



Fabrik der Zukunft — Episode 1^{V2}

Dokumentation, Spezifikation, Konstruktion

Prof. Dr.-Ing. V.Plenk Prof. Dr. rer. nat P. Stöhr
+ siehe Projekt-Handbuch-Historie*

V2.4 (PDF-Revision 1.11) (Edit Plenk 2.11.2016)

Abstract

Dieses Dokument dient als Container für die im Lauf der Arbeiten an der „Fabrik der Zukunft“ entstehenden Unterlagen. Die Quellen für dieses Dokument liegen auf dem Subversion-Server (http://svn-serv.fh-hof.de/viewvc/fdz_episode1V2/trunk).

Es soll von den bearbeitenden Gruppen parallel zu den Entwicklungsarbeiten weiterentwickelt werden. Eventuell notwendige Zugangsberechtigungen für den Subversion-Server vergibt Herr Ott (jott@fh-hof.de).

Momentan umfasst das Dokument die komplette Neuspezifikation aus dem SS2005, die Ergebnisse der Arbeiten aus dem SS2006 (Lagersystem), die Ergebnisse aus dem Praxisblock im

*http://svn-serv.fh-hof.de/viewvc/fdz_episode1V2/trunk/binaries/docs/Projekt.pdf?view=co

August 2006 (Robotersystem), die Ergebnisse aus dem WS2006/2007 (Steuerung) und die Ergebnisse der Arbeiten in Rechnergesteuerte Anlagen II aus dem SS2007 (Transportsystem, E/A-System).

Inhaltsverzeichnis

Teil I

Spezifikation

1 Einführung

1.1 Struktur der Anlage

Dieser Abschnitt soll einen groben Überblick über das Gesamtsystem liefern und die Verknüpfung der einzelnen Teile miteinander verdeutlichen. Die nähere Spezifikation des jeweiligen Moduls / Subsystems findet man in den betreffenden Kapiteln vor.

Zuerst ist es jedoch von Vorteil, die einzelnen Teile und Module aufzuzeigen:

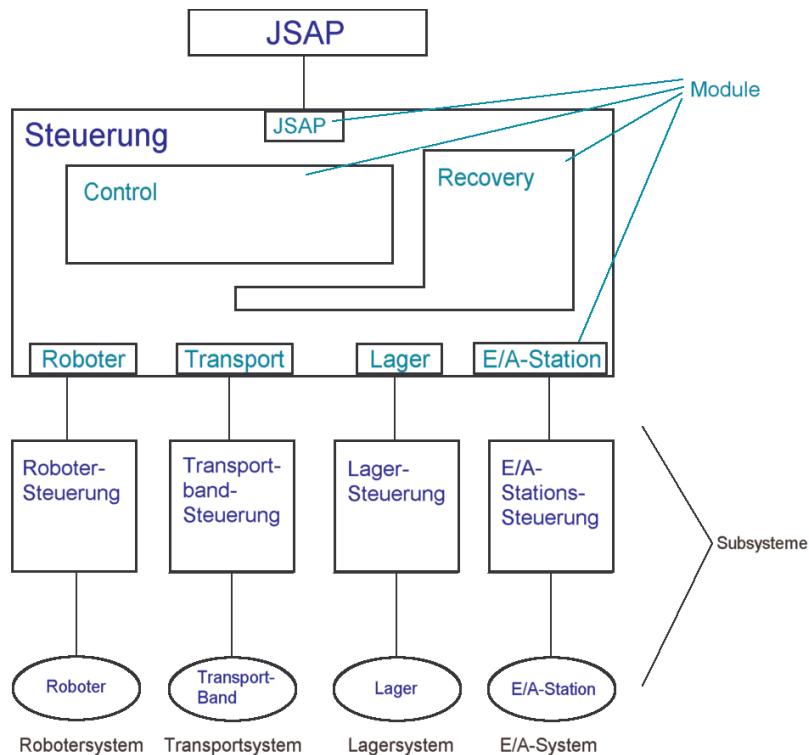


Abbildung 1.1: Systemübersicht

1.1.1 JSAP

JSAP stellt die Schnittstelle zwischen der Steuerung und einem SAP R3/System dar. Über diese Schnittstelle werden Nachrichten ausgetauscht. Damit ist es beispielsweise möglich, vom R3/System aus Produktionsaufträge an die Steuerung zu senden. JSAP hat dabei die Aufgabe, die entsprechenden Nachrichten so umzuwandeln, dass die Steuerung sie verarbeiten kann.

1.1 Struktur der Anlage

Umgekehrt werden Nachrichten resp. Antworten von der Steuerung ebenfalls von JSAP konvertiert, damit sie vom R3/System weiter verarbeitet werden können.

Die Anbindung von JSAP an das R3/System erfolgt durch sogenannte Remote Function Calls. Für den Zugriff auf diese Funktionen wird die RFC-Bibliothek, auch als *libRFC* bezeichnet, von SAP zur Verfügung gestellt.

1.1.2 Die Steuerung

Dieser Teil steuert den Produktionsablauf. Dazu teilt er den von der übergeordneten Software (z.B. SAP) empfangenen Auftrag in einzelne Befehle an die untergeordneten Einheiten auf. Die Steuerung hat als eine weitere Hauptaufgabe die Verarbeitung bzw. Reaktion auf Nachrichten von den einzelnen Subsystemen (d.h. wenn ein Subsystem die Abarbeitung eines (Unter-)Auftrags meldet, muss die Steuerung dafür sorgen, dass das nächste Subsystem seine Arbeit aufnimmt).

Da die „Steuersysteme“ (z.B. das Roboterbetriebssystem) jeweils mit unterschiedlichen Programmiersprachen implementiert werden muss, ist -im Hinblick auf die spätere Erweiterbarkeit- eine Zwischenschicht eingefügt worden: die Subsystemsteuerungen. Diese Subsystemsteuerungen werden auf einem separaten Rechner und nicht direkt auf der Hardware ausgeführt. Dies erlaubt eine einfachere Realisierung des Projektes, da so ein Testsystem der „Fabrik der Zukunft“ erstellt werden kann und, bedingt durch diese Aufteilung, ein späteres Erweitern bzw. ein späterer Umbau der Anlage leichter verwirklicht werden kann.

Die Steuerung schickt jede Anweisung an das entsprechende Subsystem. Dieses Subsystem ist dann dafür verantwortlich, der jeweiligen Hardware die geeigneten Steuerbefehle zu schicken. Weiterhin führt es dazu, dass den Subsystemen ein gewisser Teil von Intelligenz überlassen wird und nicht die Steuerung jeden einzelnen Befehl schicken muss (z.B. Auftrag an Transportbandsteuerung: „brauche Schlitten x an Position y“ und nicht „fahre Band ab, halte an Stelle y“).

1.1.3 Das Robotersystem

Das Robotersystem steuert den Roboter, der Paletten vom Transportband nimmt, Paletten auf eine Arbeits- und Bestückungsposition legt und schließlich die Paletten bestückt.

1.1.4 Das Transportsystem

Das Transportsystem ist dafür verantwortlich, Schlitten mit Paletten je nach Auftrag selbstständig an eine vorgegebene Position zu fahren. Dabei gibt die Bandsteuerung die Befehle an das Transportband weiter.

1.1 Struktur der Anlage

1.1.5 Das Lagersystem

Dieser Teil lagert Palletten. Er besteht aus einem hardwarenahen Teil, der die Mechanik ansteuert, und einer übergeordneten Software, die die Lagerplätze verwaltet und die Schnittstelle zur Steuerung herstellt.

1.1.6 Das E/A-System

Das E/A-System wird momentan nur von einem PC und einem Schalter repräsentiert. Momentan wird das System noch von einem Menschen bedient, der auf Anweisung (Monitor) eine bestimmte Aktion (z.B. leere Palette von der Warteposition entnehmen oder Palette neu bestücken) durchführt. Im Hinblick auf die spätere Erweiterbarkeit wird bereits jetzt eine E/A-Stationssteuerung eingeführt, die den Steuerungen der anderen Subsysteme entspricht und demnach ebenso erweitert werden kann, speziell in dem Sinn, dass der Mensch später einmal durch einen Roboter o.ä. ersetzt werden soll / kann.

1.1.7 Beispiel eines Idealablaufs

Zur Verdeutlichung der Aufgaben der einzelnen Systeme wird hier einmal ein exemplarisches Ablauf durchgeführt:

Annahmen: Lager ist ausreichend bestückt

Lager hat jeweils rote Lagerpaletten mit mehr als 20 Schokolinsen
Es treten keine Fehler auf
Das Gesamtsystem befindet sich in seinem Wartezustand

Auftrag: Die Steuerung bekommt von dem darüberliegendem System (JSAP) den Auftrag, eine Palette mit -der Einfachheit halber 20 roten- Schokolinsen zu bestücken.

Folgende Befehle werden für die Abarbeitung verwendet:

- Abfrage bei Lagersteuerung nach 20 Smarties. Lagersteuerung bestätigt, dass 20 Smarties im Lager sind.
- Die Steuerung fordert von der Bandsteuerung einen Schlitten an der E/A-Station an. Ist dies geschehen, benachrichtigt Bandsteuerung Steuerung.
- Steuerung fordert von der E/A-Stationssteuerung eine leere Produktpalette zur Bestückung an. Ist der Befehl ausgeführt worden, bestätigt E/A-Stationssteuerung diesen bei Steuerung.
- Steuerung fordert von Bandsteuerung, den Schlitten mit der leeren Produktpalette von der E/A-Station zu Roboter zu fahren. Ist die Anforderung ausgeführt worden, wird eine Nachricht an Steuerung zurückgegeben.

1.1 Struktur der Anlage

- Steuerung schickt an die Robotersteuerung den Befehl „nimm Produktpalette“ . Ist der Befehl ausgeführt worden, bestätigt Robotersteuerung diesen bei der Steuerung.
- Der Schlitten am Roboter wird freigegeben. Dazu wird ein Befehl von Steuerung an Bandsteuerung geschickt. Hat Bandsteuerung diesen Befehl ausgeführt, benachrichtigt Bandsteuerung die Steuerung davon.
- Steuerung fordert von Bandsteuerung einen Schlitten am Lager an. Ist dies geschehen, benachrichtigt Bandsteuerung die Steuerung.
- Steuerung schickt an Lagersteuerung den Befehl: „lege rote Lagerpalette auf das Band“ . Lagersteuerung quittiert dies nach erfolgreicher Ausführung bei Steuerung.
- Steuerung fordert von Bandsteuerung, den Schlitten mit der „roten Lagerpalette“ von Lager zu Roboter zu fahren. Ist die Anforderung ausgeführt worden, wird eine Nachricht an Steuerung zurückgegeben.
- Robotersteuerung bekommt den Befehl, die „rote Lagerpalette“ vom Band zu nehmen und sie auf einem Arbeitsplatz von Roboter zu lagern. Wurde der Befehl erfolgreich ausgeführt, benachrichtigt Robotersteuerung die Steuerung.
- Der Schlitten am Roboter wird freigegeben. Dazu wird ein Befehl von Steuerung an Bandsteuerung geschickt. Hat Bandsteuerung diesen Befehl ausgeführt, benachrichtigt Bandsteuerung die Steuerung davon.
- Nun bekommt Robotersteuerung den Befehl die Produktpalette laut Matrix zu bestücken. Ist der Befehl abgearbeitet, benachrichtigt Robotersteuerung die Steuerung.
- Steuerung fordert danach von Bandsteuerung einen leeren Schlitten am Roboter an. Ist die Anforderung ausgeführt worden, wird eine Nachricht an Steuerung zurückgegeben.
- Robotersteuerung bekommt den Befehl, die bestückte Produktpalette auf das Transportband zu legen. Ist der Befehl abgearbeitet, benachrichtigt Robotersteuerung die Steuerung.
- Steuerung fordert die Bandsteuerung auf, den Schlitten mit der bestückten Produktpalette zur E/A-Station zu fahren.
- Hier angekommen, wird die E/A-Stationssteuerung von Steuerung aufgefordert, die Produktpalette auszuschleusen. Die Steuerung wartet dann auf die Rückmeldung der E/A-Stationssteuerung, dass der Befehl ausgeführt worden ist.
- Der Schlitten an der E/A-Station wird freigegeben. Steuerung schickt dazu an Bandsteuerung den „Schlittenfreigabe“ -Befehl. Hat Bandsteuerung diesen Befehl ausgeführt, benachrichtigt Bandsteuerung die Steuerung.
- Also nächstes wird Bandsteuerung benachrichtigt, dass ein leerer Schlitten zum Roboter gebracht werden muss. Wurde dies ausgeführt, wird eine Rückmeldung geschickt.
- Roboter soll nun die Lagerpalette am Arbeitsplatz zum Transportband heben. Ist dies geschehen, wird Steuerung eine Nachricht übersendet.

1.1 Struktur der Anlage

- Bandsteuerung soll nun den Schlitten mit der Lagerpalette von Roboter aus zum Lager fahren. Wenn dieser Befehl ausgeführt worden ist, wird Steuerung von Bandsteuerung benachrichtigt.
- Steuerung befiehlt Lagersteuerung, die Lagerpalette einzulagern. Nach erfolgreicher Abarbeitung bekommt Steuerung eine Nachricht von Lagersteuerung.
- Der Schlitten vor dem Lager wird nun freigegeben. Hat Bandsteuerung diese Meldung abgearbeitet, wird Steuerung davon informiert.
- Nun wird JSAP benachrichtigt, dass der Auftrag abgeschlossen wurde. Mitgegeben an diese Nachricht wird die Anzahl der verbrauchten Smarties. In diesem Fall 20 rote Smarties.

1.2 Begriffsdefinition

1.2 Begriffsdefinition

Um Verwirrungen bei der Beschreibung des Systems zu vermeiden, werden im Rahmen dieser Arbeit folgende Bezeichnungen verwendet:

Steuerung (ST)	Software, welche für die Steuerung des Gesamtsystems zuständig ist. Es tauscht die Nachrichten mit den einzelnen Subsystemsteuerungen aus. Die Steuerung erhält Aufträge von einem Systemexternen Managementsystem (JSAP).
Robotersystem	Besteht aus Robotersteuerung (Sr) und Roboter (ro)
Robotersteuerung (Sr)	Software zur Steuerung des Roboters. Kommuniziert zwischen dem Roboter und der Steuerung.
Roboter (ro)	Mechanische Einheit zur Bestückung von Paletten.
Lagersystem	Besteht aus Lagersteuerung (Sl) und Lager (la)
Lagersteuerung (Sl)	Software zur Steuerung des Lagers. Kommuniziert zwischen Lager und der Steuerung.
Lager (la)	Hardware-Lagersystem zum Einlagern der Paletten.
Transportsystem	Besteht aus Bandsteuerung (St) und Transportband (tr).
Bandsteuerung (St)	Software zur Steuerung des Transportbandes. Kommuniziert zwischen dem Transportband und der Steuerung.
TransportBand (tr)	Mechanisches Band zum Transportieren der mit Paletten bestückten oder leeren Schlitten.
E/A-System	Besteht aus E/A-Stationssteuerung (Se) und E/A-Station (ea)
E/A-Stations- steuerung (Se)	Steuerungssoftware für die E/A-Station.
E/A-Station (ea)	Einheit für die externe Ein- und Ausgabe der Schlitten bzw. Paletten am Transportband.
Schlitten/Wagen	Auf dem Transportband befindlichen Einheiten, welche zum Transportieren von Paletten dienen.
Produktpalette (PP)	Zu bestückende Palette, welche in ihrem Ausgangszustand leer ist.
Lagerpalette (LP)	Die vom Lager angeforderte Palette. Sie ist in ihrem Ausgangszustand mit Smarties gefüllt.

1.2 Begriffsdefinition

Smartie	Schokolinsen, die zum Bestücken der Paletten verwendet werden.
System	Alle Teile zusammen (Gesamtsystem).
Subsystem	Der Steuerung untergeordnetes System, welches die einzelnen Komponenten darstellt. Robotersystem, Transportsystem, Lager-system und E/A-System sind solche Subsysteme.
Subsystemsteuerung	Steuerung eines Subsystems, z.B. Robotersteuerung, Bandsteuerung, Lager-steuerung und E/A-StationsSteuerung(siehe Subsystem).
Operator	Person, welche das System bedient.
Befehl	Befehl, der von der Steuerung an ein Subsystem geschickt wird.
Elementarbefehl	Befehl, der von der Steuerung des jeweiligen Subsystems an die Hardware geschickt wird.

2 Aufgaben der Teilsysteme

2.1 Die Steuerung

Die Fabrik der Zukunft kann für den Laboraufbau als Fertigungszelle bezeichnet werden. Diese ist für die Bearbeitung eines Auftrages zur Bestückung von Produktpaletten mit Smarties eines gewünschten Musters verantwortlich. Der Teil „Steuerung“ dieser Fertigungszelle ist die zentrale Einheit. Sie steuert den gesamten Ablauf, teilt einen Auftrag in Unteraufträge, prüft deren Erfüllbarkeit, steuert den Ablauf für eine produzierbare Produktpalette und meldet eventuell nicht erfüllbare Aufträge.

Damit ist die Steuerung das zentrale Bindeglied zwischen den Subsystemen Bandsteuerung, Lagersteuerung, Robotersteuerung und E/A-Stationssteuerung und sie übernimmt die Aufgabe, alle nötigen Kommandos an das gewünschte Subsystem zu schicken.

Zusammenspiel zwischen Steuerung und den Subsystemen

Der Nachrichtenaustausch zwischen Steuerung und den Subsystemen wird durch das Client-Server-Prinzip verwirklicht. Hierbei übernimmt die Steuerung die Aufgabe des Servers und die Subsysteme die Aufgabe der Clients. Bei der Kommunikation mit dem JSAP hat das JSAP die Aufgabe des Servers und die Steuerung die Aufgabe eines Clients.

2.1.1 Abläufe der Steuerung

Die folgenden drei Abschnitte erläutern kurz die Abläufe der Steuerung.

2.1.1.1 Auftragserhalt

Die Steuerung erhält vom darüberliegenden JSAP den Arbeitsauftrag. Ein Auftrag besteht aus der Anordnung der farbigen Smarties in einer Produktpalette. Die Steuerung ist dafür verantwortlich, dass der eingegangene Arbeitsauftrag geprüft wird. Dazu zerlegt sie den Auftrag in Unteraufträge. Ein Unterauftrag besteht aus jeweils einer Farbe und deren Anordnung auf der Produktpalette. Sind die Unteraufträge erstellt, fragt die Steuerung bei der Lagersteuerung an, ob genügend Smarties der Unterauftragsfarbe vorhanden sind und somit der Unterauftrag erfüllbar ist. Wenn alle Unteraufträge erfüllbar sind, wird das JSAP von der Steuerung informiert, dass mit der Abarbeitung des Auftrages begonnen wurde. Ansonsten wird an das JSAP gemeldet, dass der Auftrag nicht bearbeitet werden kann und ein Abbruch vorgesehen ist.

2.1 Die Steuerung

2.1.1.2 Auftragsbearbeitung

Um einen Auftrag zu erfüllen, werden von der Steuerung die einzelnen Unteraufträge abgearbeitet. Dabei muss die Steuerung dafür sorgen, dass die benötigen Ressourcen für jeden Unterauftrag zum richtigen Zeitpunkt am richtigen Ort sind. Deswegen wird dieser Unterauftrag in einzelne Befehle aufgeteilt, die von den Subsystemen verstanden werden (z.B.: Fahre Schlitten zum Lager).

Diese Befehle werden sequenziell abgearbeitet. Ist das Ende erreicht, ist somit der Unterauftrag abgeschlossen und der nachfolgende Unterauftrag kann bearbeitet werden. Sind alle Unteraufträge abgearbeitet, ist der Auftrag erledigt und die Steuerung informiert das JSAP über die erfolgreiche Abarbeitung.

Sollten während der Auftragsbearbeitung leere Lagerpaletten zum Befüllen ausgeschleust worden sein, so werden diese nach dem Ausschleusen der fertigen Produktpalette befüllt und wieder zum Einlagern eingeschleust. Damit ist für diesen Fall erst der Auftrag abgeschlossen, wenn alle ausgeschleusten Lagerpaletten wieder eingeschleust wurden.

2.1.1.3 Fehlerbehandlung

Tritt in einem Subsystem ein Fehler auf, so versucht es diesen selbst zu beseitigen. Kann dieser Fehler nach mehreren Versuchen nicht vom Subsystem behoben werden, so schickt die Subsystemsteuerung an die Steuerung eine Nachricht mit einem ganz bestimmten Fehlercode und fährt sich dann herunter. Dieser Fehlercode veranlasst die Steuerung, den Befehl „Herunterfahren“ an alle übrigen Subsysteme zu schicken. Die Steuerung wartet in diesem Fall nur auf die Antwort, dass die Subsysteme den Befehl verstanden haben und fährt sich dann ebenfalls herunter. Nun kann der Fehler manuell behoben werden. Nähere Beschreibungen zur Fehlerbehandlung der Steuerung sind im Kapitel ?? nachzulesen.

2.1 Die Steuerung

2.1.2 Spezifikation der Steuerung

Die nachfolgenden Abschnitte erläutern die Tätigkeiten der Steuerung im Gesamtsystem. Dabei werden die Tätigkeiten der Steuerung nach Subsystemen aufgeführt. Beginnend mit JSAP, als Hauptinitiator für einen Auftrag, und dann mit den Subsystemen innerhalb der Fertigungszelle.

2.1.2.1 Aufgaben der Steuerung gegenüber JSAP

Dieser Abschnitt beschreibt einzeln die zu erfüllenden Aufgaben der zentralen Steuerung gegenüber dem JSAP mit dem entsprechenden Antwortverhalten.

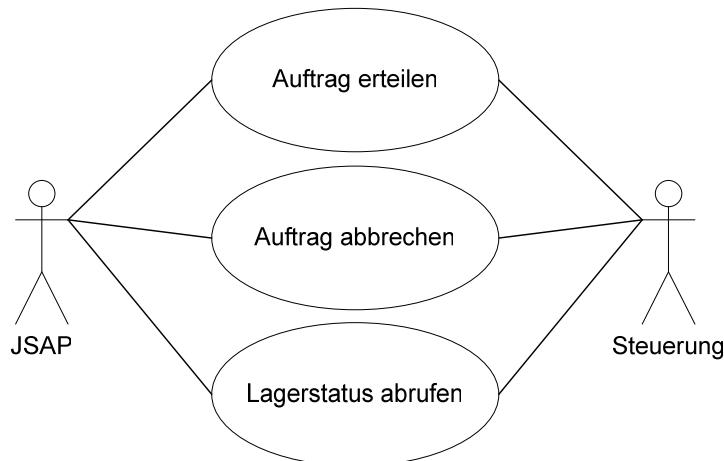


Abbildung 2.1: Befehle zwischen JSAP und der Steuerung

Definitionen:

- Es wird immer nur ein Auftrag zur Steuerung geschickt. Erst nachdem „Auftrag fertig“ an JSAP zurückgeschickt wurde, erhält die Steuerung gegebenenfalls einen neuen Auftrag.
- Der einzige Befehl bzw. die einzige Ausnahme, die die sequentielle Befehlsreihenfolge verletzen darf, ist der Befehl „Auftrag abbrechen“
 - . In diesem Fall werden von der Steuerung zwei Befehle gleichzeitig bearbeitet.
- Der Lagerstatus kann von JSAP nur angefragt werden, wenn kein Auftrag produziert wird bzw. der Auftrag fertig bearbeitet wurde und die Fertigstellung des Auftrages an das JSAP geschickt wurde.

2.1 Die Steuerung

2.1.2.1.1 Auftrag erteilen

Das JSAP teilt der Steuerung einen neuen Auftrag mit. Die Steuerung überprüft mit Hilfe der Lagersteuerung (siehe ??), ob es möglich ist, den Auftrag zu produzieren oder nicht. Das Ergebnis wird dem JSAP mitgeteilt.



Abbildung 2.2: Auftragsbestätigung an JSAP

2.1.2.1.1 Auftrag wird produziert Wenn die Lagersteuerung der Steuerung mitteilt, dass alle benötigten Smarties in der jeweils gewünschten Farbe vorhanden sind, dann wird mit der Produktion des Auftrages begonnen. Das JSAP wird darüber informiert.

2.1 Die Steuerung



Abbildung 2.3: Meldung, dass Auftrag nicht produzierbar

2.1.2.1.1.2 Auftrag kann nicht produziert werden Sollte im Lager nicht die benötigte Anzahl an Smarties in einer Farbe vorhanden sein, wird das JSAP informiert, dass es nicht möglich ist, diesen Auftrag zu produzieren. Dieser Fall sollte im Normalfall nicht vorkommen, da dem JSAP bekannt ist, wieviele Smarties in welcher Farbe im Lager vorhanden sind.

Sollte während der Produktion eines Auftrages ein neuer Auftrag zur Steuerung geschickt werden, dann wird dieser mit der Meldung „Auftrag kann nicht produziert werden“ abgelehnt.

2.1 Die Steuerung

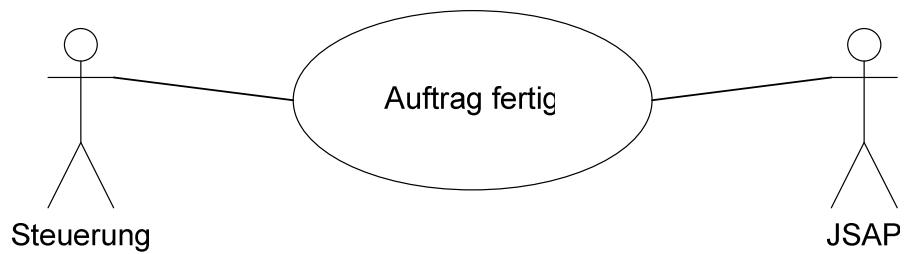


Abbildung 2.4: Fertigstellung des Auftrages melden

2.1.2.1.1.3 Auftrag fertig Wurde die Produktpalette ausgeschleust und das Gesamtsystem in den Wartezustand gebracht, dann wird dem JSAP mitgeteilt, dass der Auftrag fertig ist.

2.1 Die Steuerung



Abbildung 2.5: Produktionsfehler melden an JSAP

2.1.2.1.1.4 Fehler während der Produktion aufgetreten. Sollte es Probleme während der Produktion geben, die die Steuerung nicht selbstständig lösen kann, wird dies dem JSAP mitgeteilt. Dies könnte z.B. der Fall sein, wenn eine Komponente des Gesamtsystems während der Produktion komplett ausfällt und alle anderen Komponenten heruntergefahren werden müssen.

2.1 Die Steuerung

2.1.2.1.2 Auftrag abbrechen

Wird von JSAP der Befehl „Auftrag abbrechen“ an die Steuerung geschickt, so beendet die Steuerung die Produktion eines Auftrages zum nächstmöglichen Zeitpunkt. Das bedeutet, die Steuerung beendet einen in Produktion befindlichen Unterauftrag und löscht alle weiteren anstehenden Unteraufträge.



Abbildung 2.6: Auftragsabbruch melden an JSAP

2.1.2.1.2 Auftrag abgebrochen Die Grafik ?? zeigt, dass die Steuerung eine Antwort an das JSAP auf den Befehl „Auftrag abbrechen“ schickt. Im Kapitel ?? wird näher erläutert, wie der Ablauf ist. Die Steuerung meldet, dass sie den Befehl verstanden hat und löscht alle Unteraufträge, die noch zu bearbeiten wären. Ist dies geschehen, sendet sie ein „Ausgeführt“ an das JSAP. Wurde der in Produktion befindliche Unterauftrag abgeschlossen, so veranlasst die Steuerung das Ausschleusen der Produktpalette und das System wird in den Wartezustand gebracht. Das JSAP wird mit einem „Auftrag abgeschlossen“ über den Abschluss der Arbeiten informiert.

2.1 Die Steuerung

2.1.2.1.3 Lagerstatus abrufen

JSAP benötigt für seine Auftragsplanung die Möglichkeit, die Anzahl und Farbe der sich im Lager befindlichen Smarties abzurufen. Diese Anfrage kann allerdings erst an Steuerung gestellt werden, wenn kein Auftrag produziert wird und sich das System im Wartezustand befindet. Um diese Anfrage beantworten zu können, erfragt Steuerung bei der Lagersteuerung von jeder vorhandenen Farbe die Anzahl der zur Zeit im Lager befindlichen Smarties.



Abbildung 2.7: Lagerstatus an JSAP melden

2.1.2.1.3.1 Lagerstatus zurückgeben JSAP wird anschließend darüber informiert, wieviele Smarties je Farbe derzeit im Lager vorhanden sind.

2.1 Die Steuerung

2.1.2.2 Robotersteuerung

Für die Kommunikation mit der Robotersteuerung werden zwei Fälle unterschieden. Der eine Fall ist die Behandlung einer Lagerpalette.

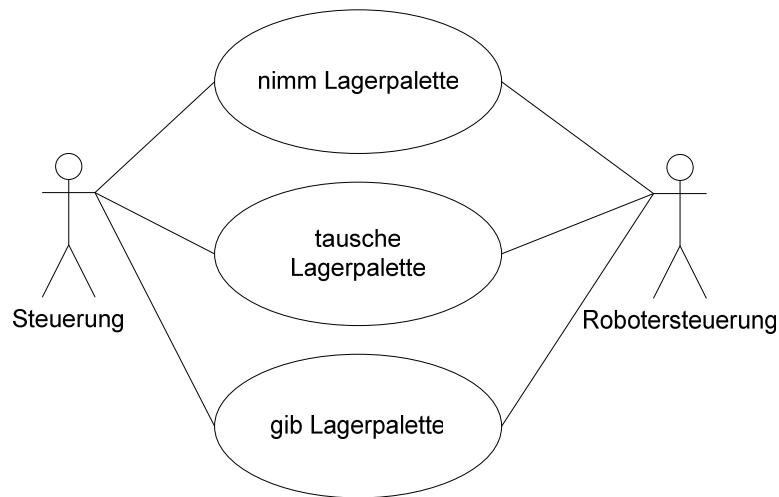


Abbildung 2.8: Kommunikation mit der Robotersteuerung - Behandlung einer Lagerpalette -

Der andere Fall: die Behandlung einer Produktpalette.

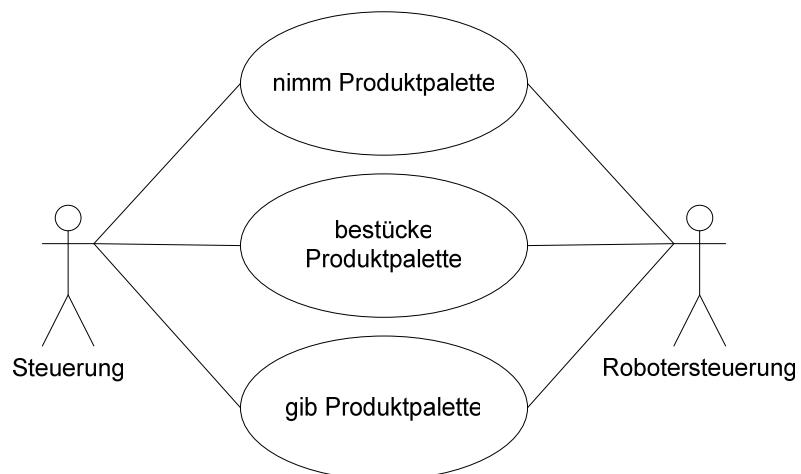


Abbildung 2.9: Kommunikation mit der Robotersteuerung - Behandlung einer Produktpalette -

2.1 Die Steuerung

Wie bei allen Subsystemen, gibt es auch in der Kommunikation zwischen Steuerung und Robotersteuerung den Befehl „Herunterfahren“ (siehe ??).

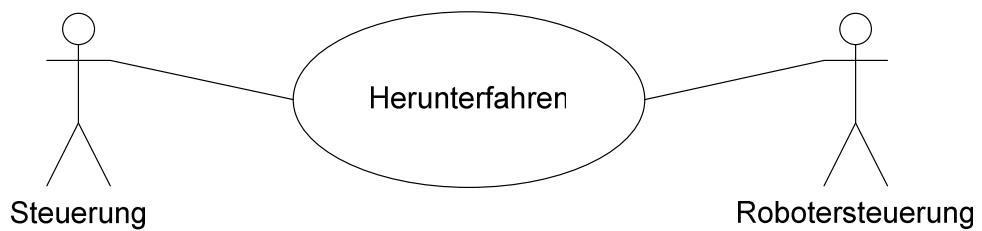


Abbildung 2.10: Kommunikation mit der Robotersteuerung - Herunterfahren -

Nähere Beschreibungen über den Roboter und die Robotersteuerung sind im Kapitel ?? nachzulesen.

2.1 Die Steuerung

2.1.2.3 Lagersteuerung

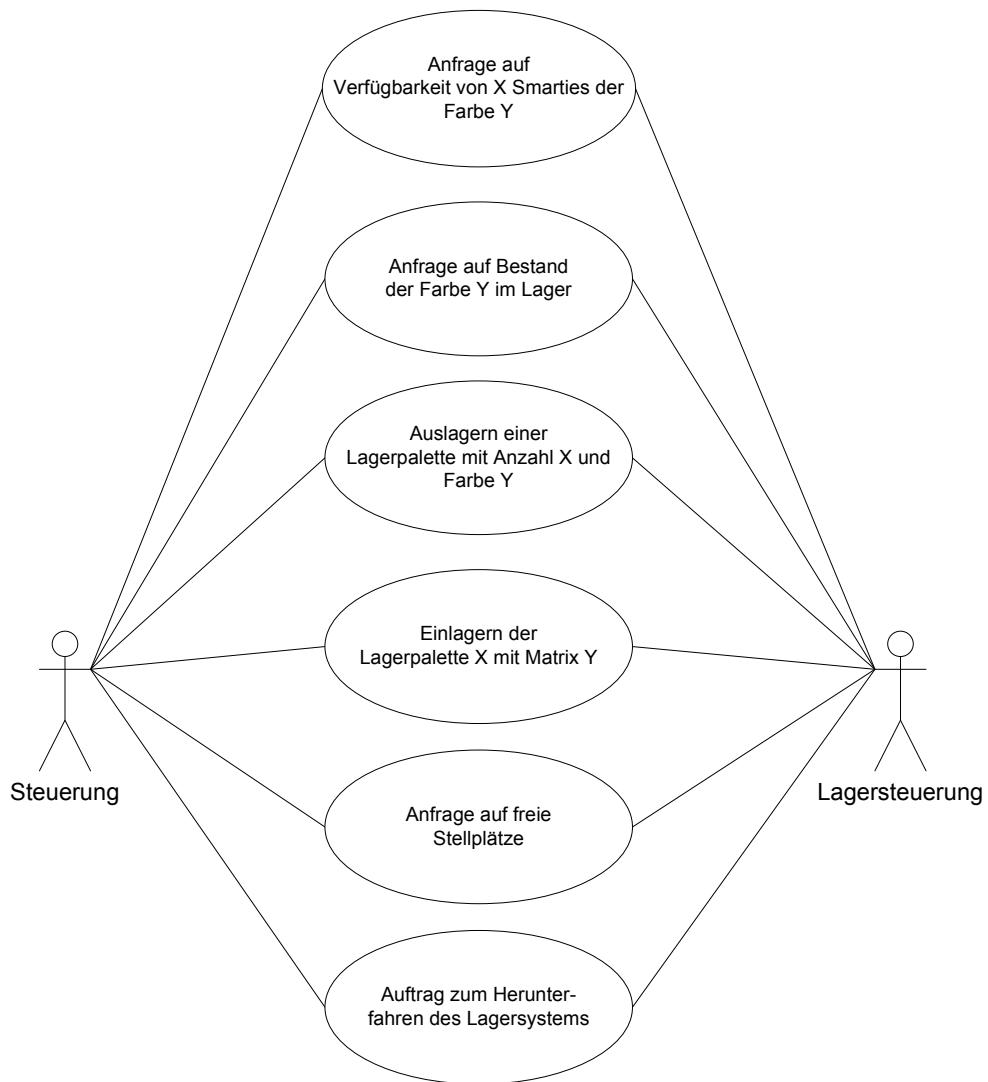


Abbildung 2.11: Kommunikation mit der Lagersteuerung

Die Befehle, die die Steuerung an die Lagersteuerung sendet, sind in obiger Grafik dargestellt. Einzelheiten und genauere Beschreibungen sind im Kapitel ?? über die Lagersteuerung und das Lager nachzulesen.

2.1 Die Steuerung

2.1.2.4 Bandsteuerung

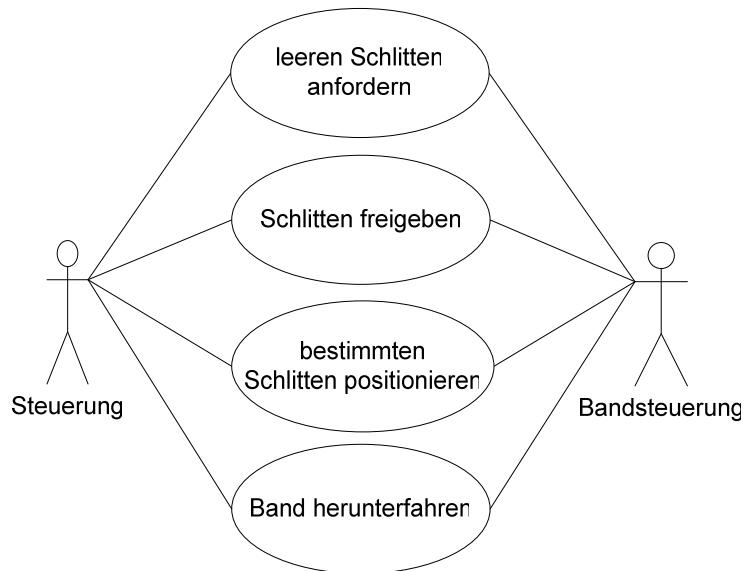


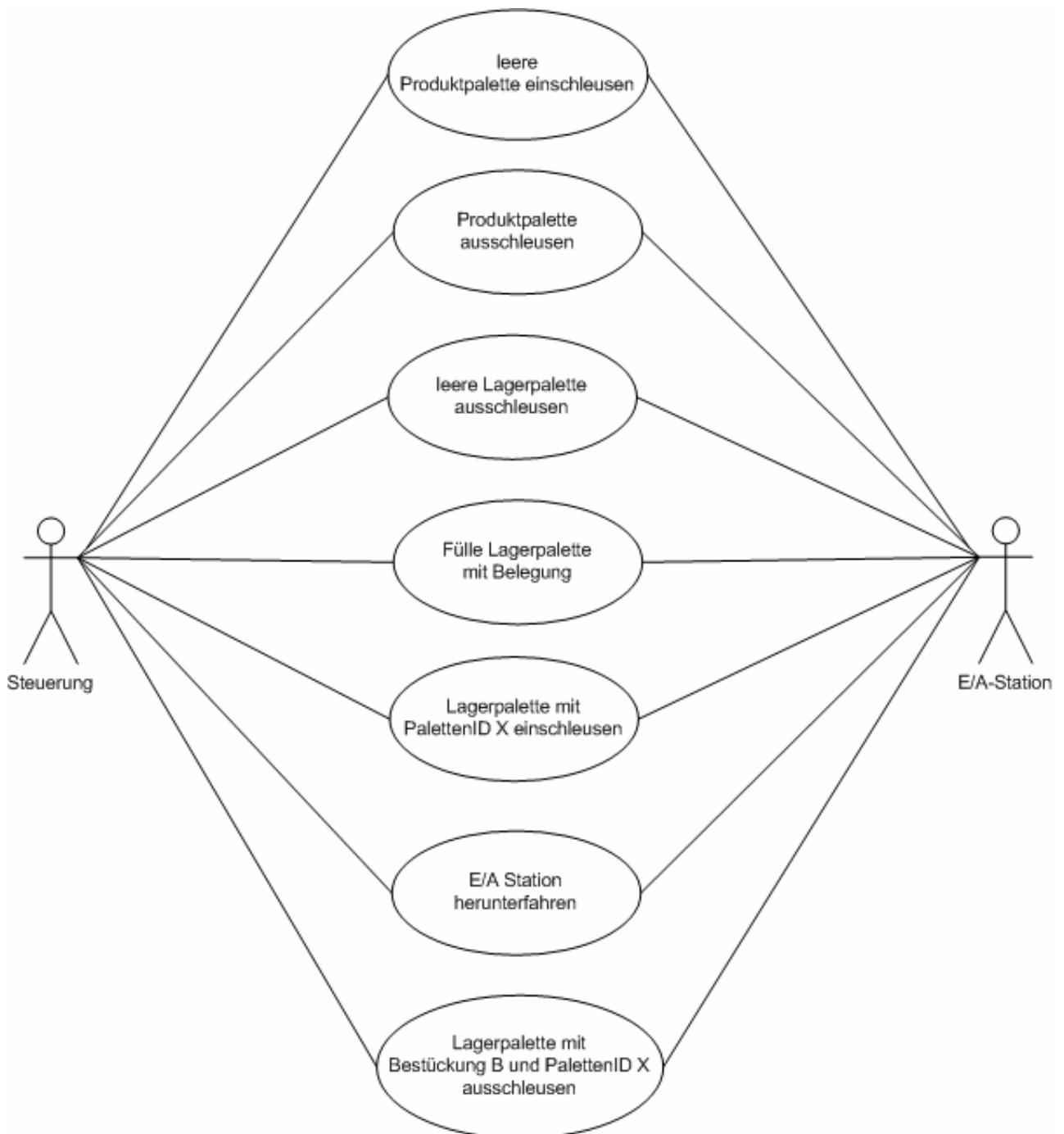
Abbildung 2.12: Kommunikation mit der Bandsteuerung

Die Grafik zeigt, welche möglichen Kommandos an die Bandsteuerung geschickt werden können. Nähere Beschreibungen zu den einzelnen Kommandos sind im Kapitel ?? über die Bandsteuerung nachzulesen.

2.1 Die Steuerung

2.1 Die Steuerung

2.1.2.5 E/A-Stationensteuerung



2.1 Die Steuerung

Die Kommunikation mit der E/A-Stationssteuerung ist im Kapitel ?? über die E/A-Stationssteuerung näher beschrieben. Diese Grafik zeigt, welche möglichen Kommandos die Steuerung an die E/A-Stationssteuerung schicken kann.

2.2 Das Robotersystem

2.2 Das Robotersystem

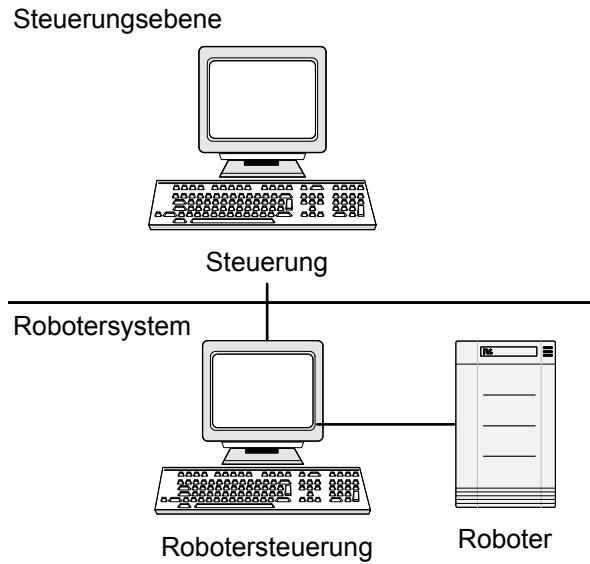


Abbildung 2.14: Aufbau des Robotersystems

2.2.1 Einführung

Das Robotersystem der Fabrik der Zukunft besteht aus der übergeordneten Robotersteuerung und einem neuen Sechsachsroboter (TX60L) der Firma Stäubli. Der Robotercontroller hat ein eigenes objektorientiertes, multitaskingfähiges Betriebssystem (VAL3). Der Roboter dient dem System zum Bestücken der Produktpaletten.

2.2 Das Robotersystem

2.2.2 Aufbau

Hier der Aufbau des Systems:

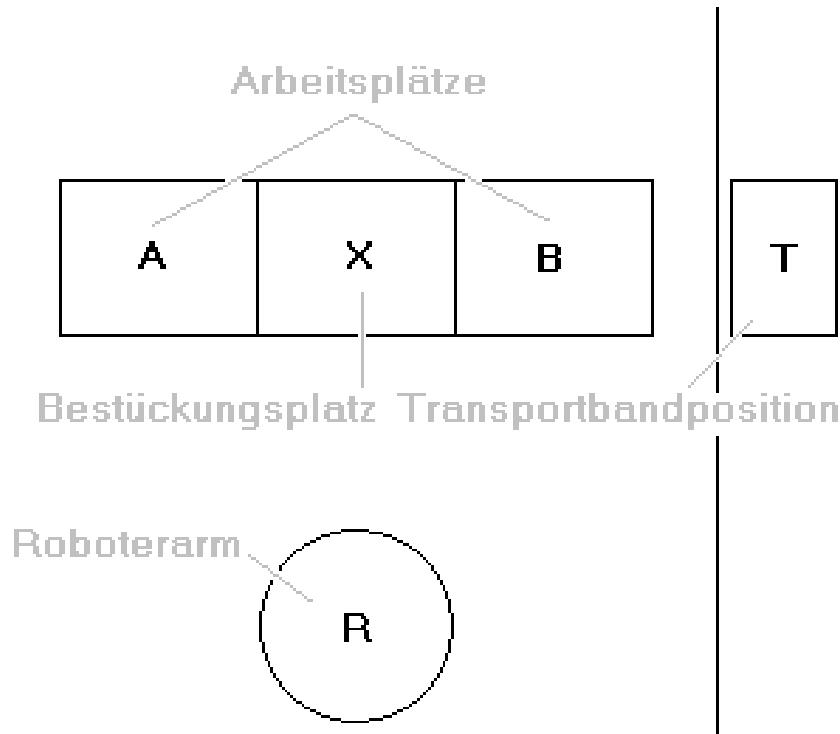


Abbildung 2.15: Aufbau

Im folgenden werden in den Texten bzgl. des Roboters weitestgehend die Abkürzungen der einzelnen Elemente des Roboteraufbaues verwendet.

In der Zeichnung beschreibt R den festmontierten Roboterarm. A und B sind die sog. Arbeitsplätze, die zur Lagerung der Lagerpaletten dienen. X, oder auch Bestückungsplatz, dient zur Lagerung einer Produktpalette, die vom Roboter mit Smarties gefüllt wird. Auf dem Transportband ist die Position T, von der Paletten geholt und Paletten zum Abtransport abgestellt werden.

Außer der Position des Roboterarms sind die genauen Positionen der Arbeits - und Bestückungsplätze nicht festgelegt, solange sich die Reihenfolge von A, X und B nicht ändert und die relativen Abstände zwischen den Systemelementen eingehalten werden. Die Reihenfolge A/X/B stellt einen optimalen Arbeitsablauf sicher, d.h., die Wege des Roboterarms sind so gering wie möglich. Zum Aufbau sollte ein Techniker oder Ingenieur zu Rate gezogen werden.

2.2.3 Details und Festlegungen

Bevor näher auf die geplante Funktionsweise des Robotersystems mit Diagrammen eingegangen wird, sollten einige Details bzgl. des Ablaufs und Aufbaus festgelegt werden:

2.2 Das Robotersystem

- Nach einem harten Systemfehler (siehe Fehlerantworten, Sequenzdiagramme und Fehlerfälle) darf der Roboter nicht ohne menschlichen Eingriff wieder die Arbeit aufnehmen.
- Die Robotersteuerung merkt sich die Paletteninformationen¹ der abgestellten Paletten.
- Dem Roboterbetriebssystem VAL3 ist so viel Arbeit und Wissen über das System wie möglich zu entziehen. Umso weniger Fehler können dort auftreten.
(Single-Point-Of-Failure)
- Die Robotersteuerung fungiert als Server. Der Roboter als Client.
- Die Robotersteuerung und der Roboter führen eine Auftragsspeicherung (Logfile) durch.
- Die Robotersteuerung kennt nur die aus der Zeichnung zu entnehmenden Stellplätze. Die Zuordnung der Plätze zu Koordinaten im Arbeitsbereich des Roboters geschieht durch das sogenannte „teachen“² des Roboters.
- Von den Positionen A und B ist normalerweise nur eine von beiden belegt. Beim Tauschen der Lagerpaletten sind kurzzeitig beide belegt.

2.2.4 Hinweis Robotersteuerung (Sr) an Roboter (ro)

Die Robotersteuerung (Sr) bekommt von der Steuerung (ST) einen Befehl. Siehe Abschnitt ?? für die allgemeine Beschreibung hierfür. Für den Roboter (ro) muss der Befehl allerdings umgewandelt oder aufgeteilt werden. Wir sprechen dann auch von einem Elementarbefehl und nicht mehr von einem Befehl. An sich ist das auch kein großer Aufwand, aber in diesem Abschnitt soll auf ein sehr wichtiges Detail hingewiesen werden: Nur die Robotersteuerung weiß, wo welche Palette steht und welche Paletten welche und wieviele Smarties enthalten. Deswegen ist es wichtig, sehr genau mit den Daten der gelagerten Paletten zu arbeiten. Zudem soll der Roboter auch für spätere Erweiterungen vorbereitet sein, indem nur die Robotersteuerung verändert wird. Theoretisches Beispiel: Umverteilung von Smarties einer LP auf eine andere LP.

2.2.5 Befehle Steuerung (ST) an Robotersteuerung (Sr)

nimm Lagerpalette:

¹Paletteninformation: Paletten-ID und Paletten-Matrix

²teachen: Speichern der Punkte, die der Roboter anfahren kann

2.2 Das Robotersystem

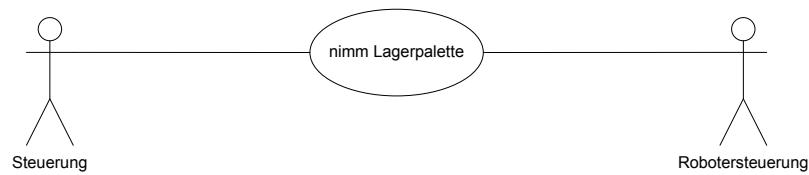


Abbildung 2.16: Nimm Lagerpalette

Anweisung, eine Lagerpalette (LP) vom Transportband (T) auf eine der beiden Positionen A oder B des Arbeitsbereiches des Roboters zu legen. Die Entscheidung auf welche Position die Palette gelegt wird trifft Robotersteuerung. Als Parameter an Sr wird die LP-Information übergeben, welche von Sr gespeichert wird. Es werden keine Informationen in der Antwort an ST zurückgegeben.

2.2 Das Robotersystem

nimm Produktpalette:

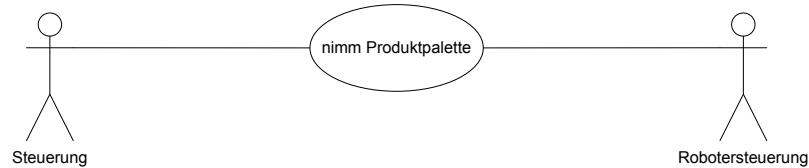


Abbildung 2.17: Nimm Produktpalette

Anweisung, die Produktpalette (PP) vom Transportband (T) auf die Bestückungsposition (X) des Arbeitsbereiches des Roboters zu legen. Sr benötigt hierfür keine Parameter. Es werden keine Informationen in der Antwort an ST zurückgegeben.

gib Produktpalette:

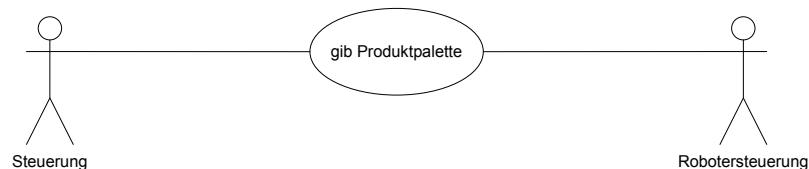


Abbildung 2.18: Gib Produktpalette

2.2 Das Robotersystem

Anweisung, die Produktpalette (PP) zurück auf das Transportband (T) zu stellen. Es werden keine Parameter an Sr übergeben und keine Informationen an ST zurückgegeben.

gib Lagerpalette:



Abbildung 2.19: Gib Lagerpalette

Anweisung, die Lagerpalette (LP) zurück auf das Transportband (T) zu stellen. Es werden keine Parameter an Sr übergeben. Sr teilt ST die LP-Information in der Antwort mit.

2.2 Das Robotersystem

fülle Produktpalette:

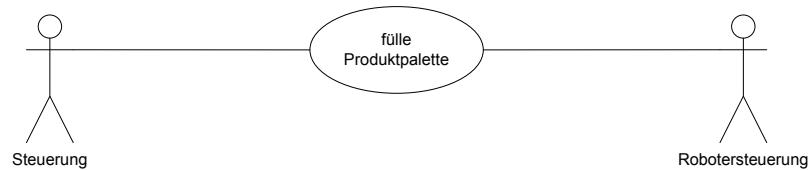


Abbildung 2.20: Fülle Produktpalette

Anweisung, die Produktpalette (PP) mit der übergebenen Produktmatrix zu bestücken. Sr teilt die übergebene Matrix in einzelne Elementarbefehle für ro auf und sendet diese an ro. Nach dem Bestücken wird an ST die abgearbeitete PP-Matrix zurückgegeben.

2.2 Das Robotersystem

tausche Lagerpaletten:

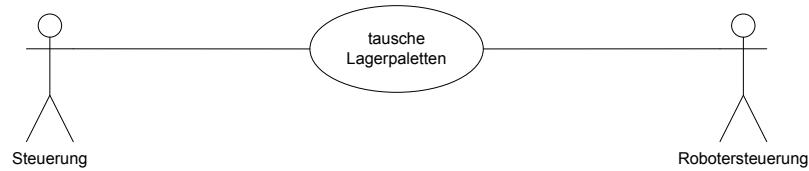


Abbildung 2.21: Tausche Lagerpaletten

Anweisung, eine Lagerpalette (LP) vom Transportband (T) auf einen freien Lagerplatz (A oder B) und die LP des anderen Lagerplatzes (A oder B) zurück aufs Transportband zu stellen. Dieser Befehl ersetzt zwei einzelne "Nimm LP" und "Gib LP" Befehle. Es kann immer mit diesem Befehl gearbeitet werden, da Sr weiß ob und wo bereits eine LP steht. Nur zum kompletten Aufräumen des Arbeitsplatzes muss ein einzelner "Gib LP" Befehl gesendet werden. Als Parameter an Sr wird die LP-Information der neuen LP auf dem Transportband (T) übergeben. Als Antwort sendet Sr an ST die LP-Information der LP, die von Sr auf das Transportband gestellt wurde.

2.2 Das Robotersystem

Reset/Anfangszustand (Herunterfahren):

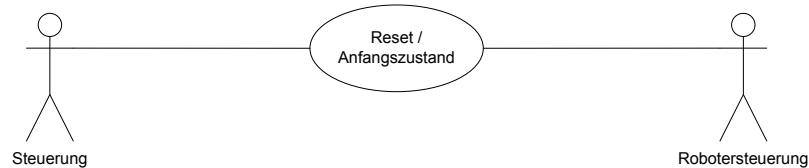


Abbildung 2.22: Reset/Anfangszustand

Anweisung, Sr und ro von einem definierten Zustand in den Anfangszustand zurückzusetzen. Sr sendet A001 und A002 an die ST.

2.2.6 Elementarbefehle Robotersteuerung (Sr) an Roboter (ro)

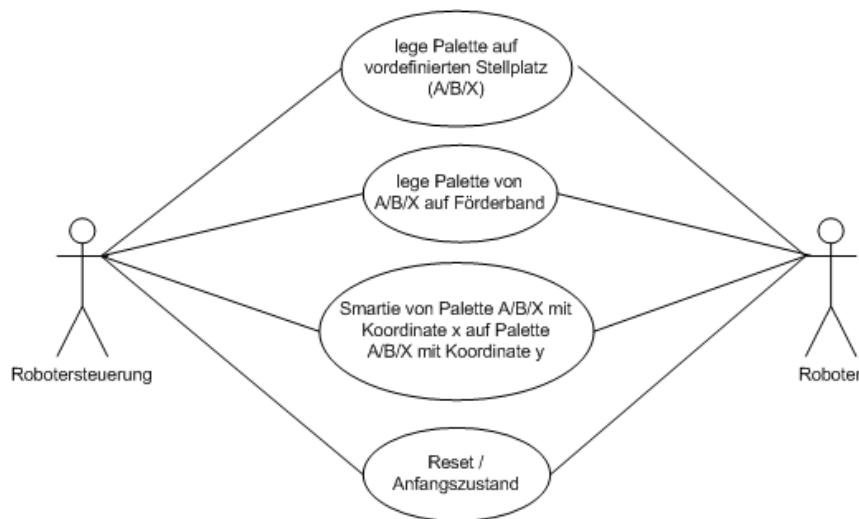


Abbildung 2.23: Robotersteuerung an Roboter

lege Palette auf vordefinierten Stellplatz:

2.2 Das Robotersystem

Anweisung, eine Palette vom Transportband (T) auf eine der 3 Positionen auf dem Arbeitsplatz zu legen. Als Parameter an ro kann A, B oder X übergeben werden. Es werden keine Informationen an Sr zurückgegeben.

lege Palette von Stellplatz auf Transportband:

Anweisung, eine Palette von einer der 3 Positionen auf dem Arbeitsplatz auf das Transportband (T) zu legen. Als Parameter an ro kann A, B oder X übergeben werden. Es werden keine Informationen an Sr zurückgegeben.

befördere ein Smartie:

Anweisung, ein Smartie <Index> der Palette <Von A/B/X> auf die Palette <Nach A/B/X> an die Stelle <Index> zu legen. Es werden keine Informationen von ro an Sr zurückgegeben.

Reset/Anfangszustand (Herunterfahren):

Anweisung, ro in seine Ausgangsposition zurückzufahren. Es werden keine Parameter an ro übergeben und Sr erwartet A001 vom Roboter.

2.2.7 Antworten Robotersteuerung (Sr) an Steuerung (ST)

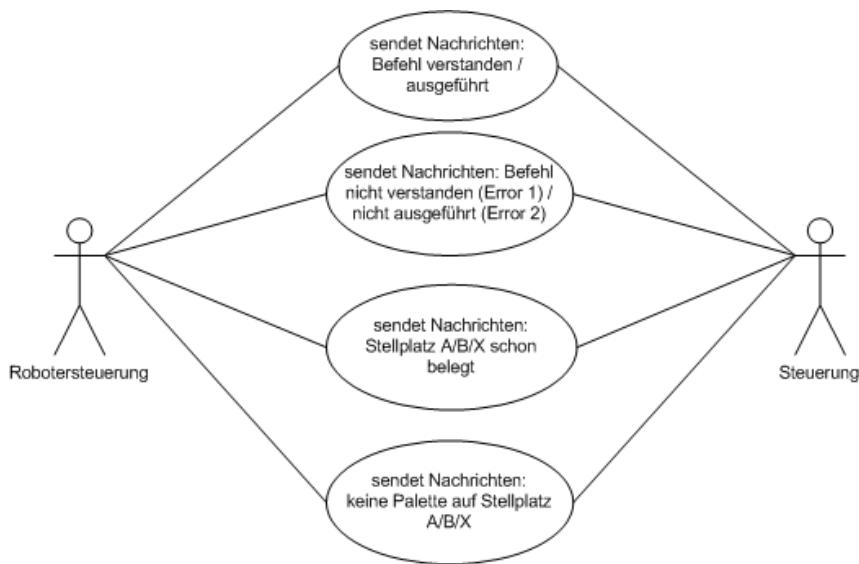


Abbildung 2.24: Robotersteuerung an Steuerung

Nachricht verstanden:

Befehl wurde empfangen und verstanden. Der Timecode des Ursprungsbefehls wird unverändert mit zurückgegeben.

Nachricht ausgeführt. Hier gibt es 5 verschiedene Antworten, je nach Befehl:

2.2 Das Robotersystem

Befehl wurde erfolgreich ausgeführt. Der Timecode des Ursprungsbefehls wird unverändert mit zurückgegeben. Dies ist die Standardantwort auf alle Befehle, bis auf die folgenden Ausnahmen.

Nach Befehl die LP auf das Band zu stellen oder die LP zu tauschen:

Befehl wurde erfolgreich ausgeführt. Der Timecode des Ursprungsbefehls wird unverändert mit zurückgegeben. Zusätzlich wird die Paletteninformation der LP mit an Steuerung übergeben.

Nach Befehl die PP zu bestücken

Befehl wurde erfolgreich ausgeführt. Der Timecode des Ursprungsbefehls wird unverändert mit zurückgegeben. Zusätzlich wird die Bestückungsmatrix zurückgegeben.

Befehl wurde nicht verstanden

z.B. ungültiger Befehl, Netzwerk-Übertragungsfehler

Befehl konnte nicht ausgeführt werden

Der Befehl konnte nicht ausgeführt werden, z.B. Netzwerk Problem zwischen Robotersteuerung <-> Roboter, Roboter reagiert nicht usw.

Keine Lagerpalette auf A/B vorhanden

Auf den Lagerpositionen A/B befindet sich keine Palette.

Position für Lagerpalette bereits belegt

Auf der Lagerposition befindet sich bereits eine Lagerpalette

Position für Produktpalette bereits belegt

Auf der Position für die Produktpalette befindet sich bereits eine Produktpalette

Position auf Produktpalette bereits belegt

Auf mind. einer Produktpaletten-Position, welche durch den Bestückungsauftrag belegt werden soll, befindet sich bereits ein Smartie

Schwerer Fehler im System. Ganzes System soll/muss angehalten werden.

Wenn Roboter nicht reagiert o.ä., wird F999 gesendet, woraufhin das gesamte System anhalten soll/muss.

2.2 Das Robotersystem

2.2.8 Antworten Roboter (ro) an Robotersteuerung (Sr)



Abbildung 2.25: Roboter an Robotersteuerung

Elementarbefehl wurde verstanden und wird nun ausgef\u00fchrt

Elementarbefehl wurde verstanden, der Timecode des Ursprungsbefehls wird unver\u00e4ndert mit zur\u00fcckgegeben.

Elementarbefehl wurde erfolgreich ausgef\u00fchrt.

Elementarbefehl wurde erfolgreich ausgef\u00fchrt der Timecode des Ursprungsbefehls wird unver\u00e4ndert mit zur\u00fcckgegeben.

Elementarbefehl wurde nicht verstanden

Der Elementarbefehl wurde nicht verstanden. z.B. ung\u00fcltiger Elementarbefehl oder Netzwerkunterbrechung. Der Timecode des Ursprungsbefehls wird unver\u00e4ndert mit zur\u00fcckgegeben.

Elementarbefehl konnte nicht ausgef\u00fchrt werden

Der Elementarbefehl konnte nicht ausgef\u00fchrt werden. z.B. mechanisches Problem beim Roboter. Der Timecode des Ursprungsbefehls wird unver\u00e4ndert mit zur\u00fcckgegeben.

2.3 Das Lagersystem

2.3.1 Aufbau des Lagersystems

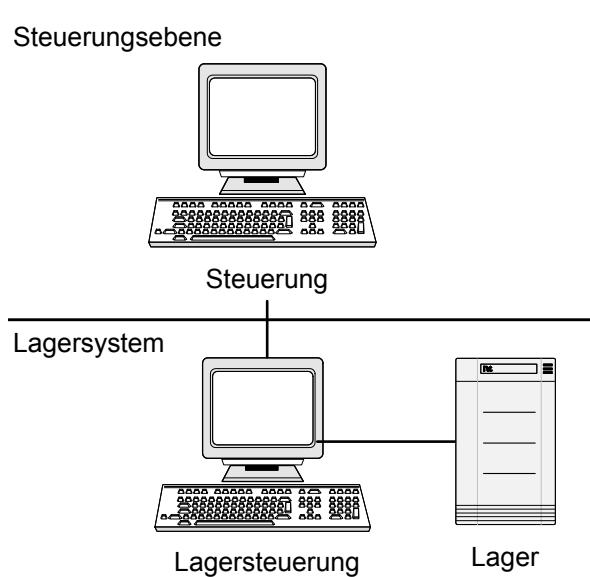


Abbildung 2.26: Aufbau Lagersystem

Das Lagersystem besteht aus dem physikalischen Lager und der Lagersteuerung. Die Mechanik des Lagers wird über einen Mikroprozessor gesteuert, welcher mit der Lagersteuerung kommuniziert. Die Mechanik ermöglicht das Ein- und Auslagern von Lagerpaletten zwischen dem Transportschlitten und den Lagerfächern. Es gibt zwei Arten von Paletten. Eine Lagerpalette enthält 91 Smartieslots im Aufbau 7x13, eine Produktpalette 63 Smartieslots im Aufbau 7x9. Zur Verfügung stehen 15 Fächer. Die Lagersteuerung ist mit der zentralen Steuerung verbunden und verwaltet den Bestand und die Lagerplätze innerhalb des Lagers.

2.3 Das Lagersystem

2.3.2 Aufgaben innerhalb des Gesamtsystems

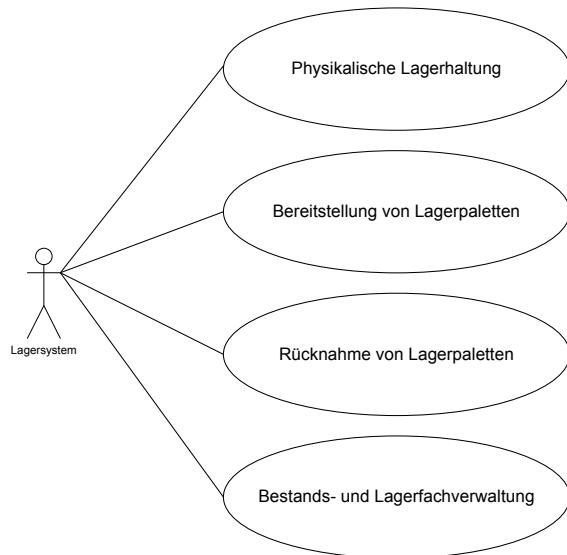


Abbildung 2.27: Aufgaben des Lagersystems

Das Lagersystem ist dafür zuständig, Smartie-Paletten mit jeweils einer bestimmten Farbe zu lagern und dem Transportsystem die zur Abarbeitung des Auftrags benötigten Lagerpaletten zu übergeben. Nachdem die ausgelagerte Lagerpalette zum Roboter transportiert und der Bestückungsvorgang vom Roboter abgeschlossen wurde, befördert das Transportsystem die Lagerpalette zurück zum Lager. Das Lagersystem ist hier wiederum für die korrekte Einlagerung verantwortlich. Sollte eine Lagerpalette während des Bestückungsvorgangs leer werden, kommt diese nicht zurück ins Lager. Stattdessen wird sie an der EA-Station ausgeschleust. Erst wenn die Lagerpalette aufgefüllt und wieder eingeschleust wurde, kann sie zurück ins Lager. Die Lagerpalette kann beim Auffüllen mit einer neuen Farbe bestückt werden.

Des weiteren ist das Lagersystem für die Bestandsführung verantwortlich. Die Lagersteuerung muss jederzeit wissen, wie viele Smarties einer bestimmten Farbe im Lager vorhanden sind und auf welchen Lagerpaletten sie sich befinden. Dazu wird für jede Lagerpalette die Belegung der Matrix gespeichert.

2.3 Das Lagersystem

2.3.3 Rahmenbedingungen und Vereinbarungen

Die folgenden Festlegungen wurden innerhalb der gesamten Projektgruppe FDZ für den Teil Lagersystem und allen damit zusammenhängenden Komponenten erörtert und schließlich festgelegt:

- Im Lager stehen ausschließlich mit mindestens einem Smartie bestückte Lagerpaletten, d.h. keine leeren Lagerpaletten
- Leere Lagerpaletten werden an der E/A-Station ausgeschleust und u.U. wieder mit einer von der Steuerung vorgesehenen Farbe befüllt
- Lagerpaletten sind einfarbig belegt
- Lagerpaletten werden nur solange datentechnisch und logisch vom Lagersystem verwaltet, solange sie sich physikalisch innerhalb des Lagers befinden
- Lagerpaletten haben innerhalb des Lagers keinen festen Standort, d.h. bei Einlagerung einer Lagerpalette, die bereits einmal im Lager gebucht war, kann diese einem völlig anderen Stellplatz zugewiesen werden als bei der letzten Buchung
- Lagerpaletten können in beliebiger Reihenfolge in das Lager ein- und ausgeschleust werden
- Im Lager stehen zu keinem Zeitpunkt leere oder gefüllte Produktpaletten
- Leere Produktpaletten werden über die E/A-Station eingeschleust und ohne Mitwirken der Lagersteuerung direkt zum Roboter befördert
- Bei unplanmäßigem Beenden des Lagersystems wird beim anschließendem Neustart der Lagersteuerungssoftware versucht, u.U. mit Hilfe des Lageroperators den letzten bekannten Auftrag erneut auszuführen (siehe ?? Systemstart und Recovery-Modus)

2.3 Das Lagersystem

2.3.4 Aufgaben des Lagers gegenüber der Lagersteuerung

2.3.4.1 Einlagerung einer Lagerpalette

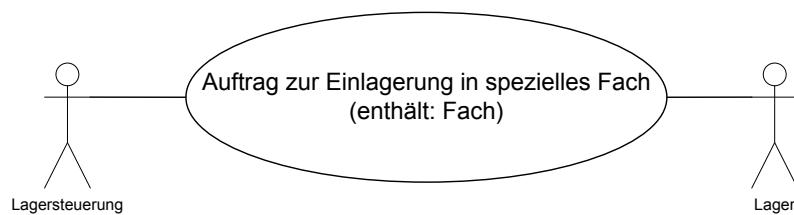


Abbildung 2.28: Auftrag von der Lagersteuerung an das Lager zur Einlagerung einer Lagerpalette in Fach X

Nachdem die Lagersteuerung von der Steuerung den Befehl zum Einlagern einer Lagerpalette bekommen und ein leeres Fach ermittelt hat, sendet die Lagersteuerung den Elementarbefehl zum Einlagern der Lagerpalette an das Lager. Dabei wird das ermittelte Einlagerfach als Parameter übergeben. Das Lager holt sich dann die Lagerpalette vom Schlitten des Transportbands und befördert sie in das übergebene Fach.

2.3 Das Lagersystem

2.3.4.1.1 Einlagervorgang erfolgreich abgeschlossen

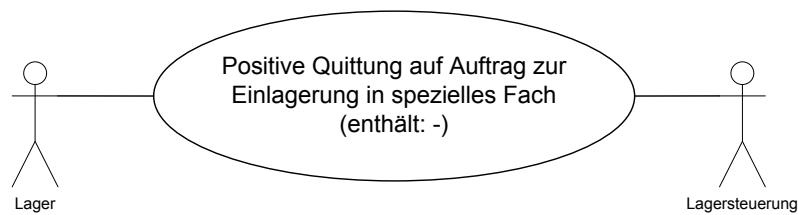


Abbildung 2.29: positive Quittung für Auftrag zur Einlagerung einer Lagerpalette in Fach X

Wenn das Einlagern erfolgreich war, meldet das Lager den Vollzug des Einlagervorgangs an die Lagersteuerung. Diese bucht die Lagerpalette in den Datenbestand.

2.3 Das Lagersystem

2.3.4.1.2 Einlagervorgang nicht erfolgreich abgeschlossen

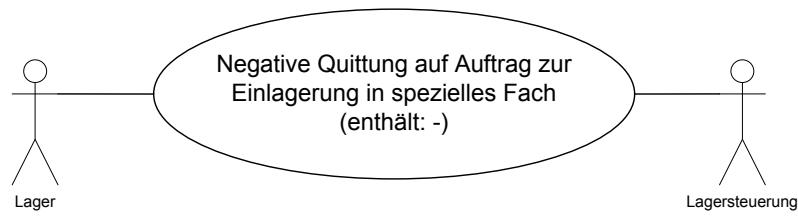


Abbildung 2.30: negative Quittierung für Auftrag zur Einlagerung einer Lagerpalette in Fach X

Falls die Einlagerung aufgrund eines mechanischen Defekts o.ä. fehlschlägt, gibt das Lager eine Fehlermeldung an die Lagersteuerung zurück. An dieser Stelle soll dem Lageroperator ermöglicht werden, den Fehler soweit möglich zu beheben und den Vorgang im Lager zu wiederholen. Schlägt dieser z.B. wegen eines schwerwiegenden Fehlers wieder fehl, kann der Lageroperator den Vorgang abbrechen. In diesem Fall reicht die Lagersteuerung die Fehlermeldung an die Steuerung entsprechend weiter.

2.3 Das Lagersystem

2.3.4.2 Auslagern einer Lagerpalette

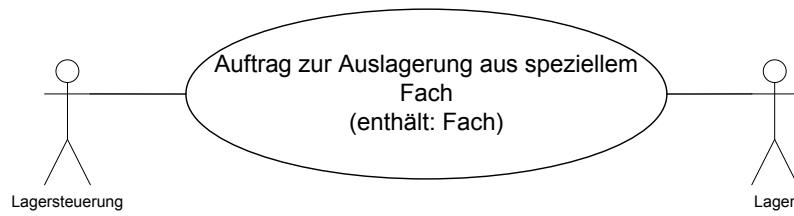


Abbildung 2.31: Auftrag von der Lagersteuerung an das Lager zur Auslagerung einer Lagerpalette aus Fach X

Beim Auslagern einer Lagerpalette sendet die Lagersteuerung den Elementarbefehl, eine bestimmte Lagerpalette aus dem Fach x auszulagern. Das Lager holt dann die Lagerpalette aus dem Palettenschacht und schiebt sie auf den Schlitten des Transportbands.

2.3 Das Lagersystem

2.3.4.2.1 Auslagervorgang erfolgreich abgeschlossen

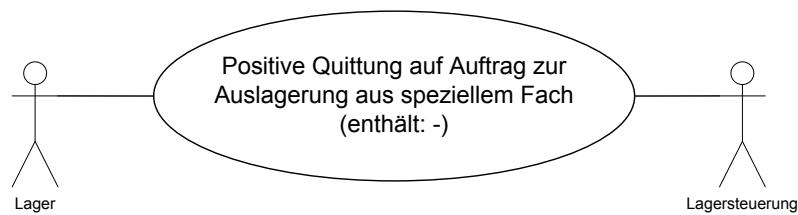


Abbildung 2.32: positive Quittung für Auftrag zur Auslagerung einer Lagerpalette aus Fach X

Wenn die Auslagerung erfolgreich war, meldet das Lager den Vollzug der Auslagerung an die Lagersteuerung. Diese bucht die Lagerpalette aus dem gegenwärtigen Bestand.

2.3 Das Lagersystem

2.3.4.2.2 Auslagervorgang nicht erfolgreich abgeschlossen

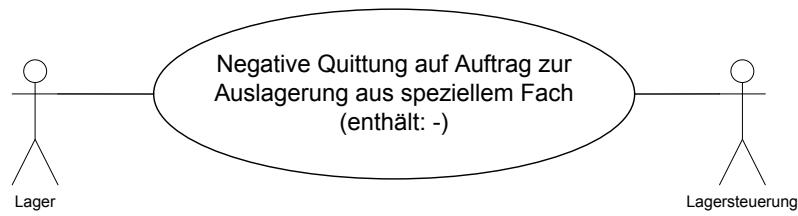


Abbildung 2.33: negative Quittung für Auftrag zur Auslagerung einer Lagerpalette aus Fach X

Sollte die Auslieferung der Lagerpalette nicht erfolgreich durchgeführt worden sein, so schickt das Lager eine negative Quittung an die Lagersteuerung. An dieser Stelle soll dem Lageroperator ermöglicht werden, den Fehler soweit möglich zu beheben und den Vorgang im Lager zu wiederholen. Schlägt dieser z.B. wegen eines schwerwiegenden Fehlers wieder fehl, kann der Lageroperator den Vorgang abbrechen. In diesem Fall reicht die Lagersteuerung die Fehlermeldung an die Steuerung entsprechend weiter.

2.3.5 Aufgaben der Lagersteuerung gegenüber der zentralen Steuerung

Dieses Kapitel beschreibt einzeln die zu erfüllenden Aufgaben gegenüber der zentralen Steuerung mit entsprechendem Antwortverhalten.

2.3 Das Lagersystem

2.3.5.1 Anmeldung an Steuerung

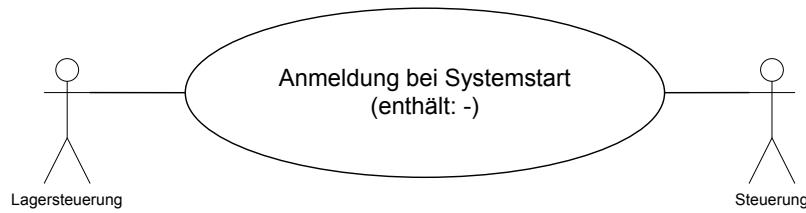


Abbildung 2.34: Anmeldung an Steuerung bei Systemstart

Die Anmeldung an der zentralen Steuerung erfolgt am Ende der Initialisierung des Lagersystems. Die Initialisierung wird immer beim Hochfahren des Lagersystems vorgenommen. Sie beinhaltet das Einlesen des zuletzt bekannten Lagerzustandes. Dieser enthält Position und Zustand aller im Lager befindlichen Lagerpaletten, den absoluten Bestand jeder Smartiefarbe, und die Anzahl an freien Stellplätzen. Nun erfolgt die Anmeldung an der zentralen Steuerung. Weiterhin wird der Bearbeitungsstatus der letzten Anfrage an die Lagersteuerung geprüft. Wenn die letzte Anfrage vollständig bearbeitet worden ist, geht das Lagersystem in Bereitschaft. Wenn die Bearbeitung der letzten Anfrage unterbrochen wurde (Stromausfall, Systemfehler,...) versucht die Lagersteuerung, unter Miteinbeziehung des Lageroperators, die letzte Anfrage erneut auszuführen.

2.3 Das Lagersystem

2.3.5.2 Anfrage des Bestands einer Farbe

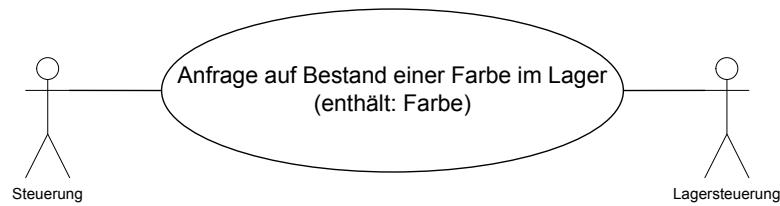


Abbildung 2.35: Anfrage des Bestands einer Farbe

Die Steuerung kann bei der Lagersteuerung die Anzahl der verfügbaren Smarties einer bestimmten Farbe im Lager nachfragen. Dies geschieht für jede Farbe, wenn das JSAP-System den Gesamtbestand des Lagers bei der Steuerung anfragt.



Abbildung 2.36: Quittung auf Anfrage des Bestands einer Farbe

2.3 Das Lagersystem

Die Lagersteuerung stellt anhand der Bestandsdatei fest, wie viele Smarties der angefragten Farbe im Lager liegen und meldet diese zurück.

2.3.5.3 Anfrage auf Verfügbarkeit einer Smartiefarbe

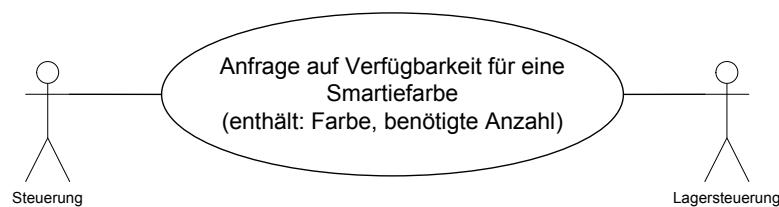


Abbildung 2.37: Anfrage auf Verfügbarkeit einer Smartiefarbe

Nach Eingang eines Auftrages in der zentralen Steuerung überprüft diese den Auftrag auf Erfüllbarkeit. Dazu fragt die Steuerung bei der LagerSteuerung nacheinander die geforderte Menge jeder im Auftrag enthaltenen Farbe nach. Aufgrund der in der Anfrage enthaltenen Farbe und Menge überprüft die Lagersteuerung den verfügbaren Bestand dieser Farbe. Verfügbar bedeutet, dass sich die Smarties physikalisch im Lager befinden müssen. Allerdings kann die Menge Smarties über mehrere Paletten verteilt sein. Die Steuerung erkennt dies bei der Auslagerung und fordert nach Bearbeitung der Palette mit unzureichendem Bestand so lange weitere Lagerpaletten mit der Restmenge an, bis die benötigte Smartieanzahl erreicht wird.

2.3 Das Lagersystem

2.3.5.3.1 Anfrage auf Verfügbarkeit einer Smartiefarbe positiv

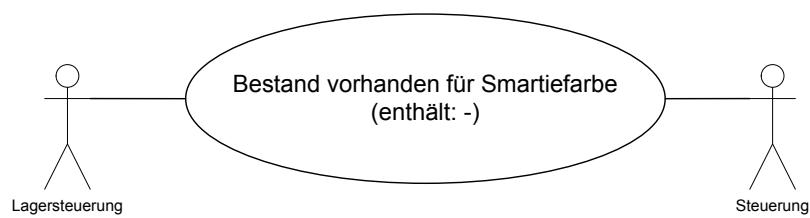


Abbildung 2.38: Anfrage auf Verfügbarkeit einer Smartiefarbe positiv quittiert

Ist die angefragte Menge verfügbar, wird eine Bestätigung an die Steuerung geschickt. Die Steuerung fragt daraufhin nacheinander weitere benötigte Farben an oder beginnt mit der Auftragsausführung, wenn alle Farben in ausreichender Anzahl vorhanden sind.

2.3 Das Lagersystem

2.3.5.3.2 Anfrage auf Verfügbarkeit einer Smartiefarbe negativ

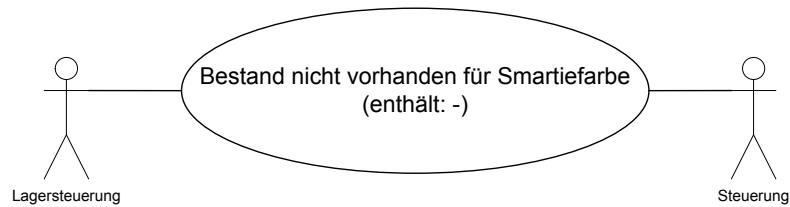


Abbildung 2.39: Anfrage auf Verfügbarkeit einer Smartiefarbe negativ quittiert

Ist die angefragte Menge nicht verfügbar, wird eine negative Quittung an die Steuerung geschickt. Daraufhin bricht die Steuerung den aktuellen Auftrag ab. Weiterhin meldet die Steuerung dem JSAP-System einen Fehler, da dieses nicht erfüllbare Aufträge nicht an die Steuerung hätte übergeben dürfen.

2.3 Das Lagersystem

2.3.5.4 Anfrage auf freie Stellplätze im Lager



Abbildung 2.40: Anfrage auf freie Stellplätze im Lager

Die Steuerung kann bei der Lagersteuerung die Anzahl der momentan leeren Stellplätze innerhalb des Lagers anfragen. Die Steuerung fragt diese nach dem Hochfahren und Anmelden aller Subsysteme und während des Betriebs nach. Diese Maßnahme soll sicherstellen, dass vor dem Anstoß einer Schrittkette zur Einlagerung einer oder mehrerer Lagerpaletten (z.B. Neubestückung des Lagers mit neuen Lagerpaletten) genügend Lagerplätze zur Ausführung der Einlagerung frei sind.

2.3 Das Lagersystem

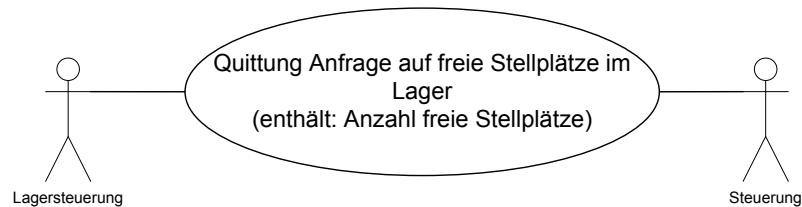


Abbildung 2.41: Antwort auf Anfrage freie Stellplätze im Lager

Die Lagersteuerung prüft daraufhin ihren Datenbestand und quittiert die Anfrage der zentralen Steuerung mit der ermittelten Anzahl freier Stellplätze.

2.3.5.5 Einlagern der Lagerpaletten



Abbildung 2.42: Einlagerauftrag von Steuerung an Lagersteuerung

2.3 Das Lagersystem

Soll eine Lagerpalette eingelagert werden, so erhält die Lagersteuerung einen entsprechenden Befehl von der zentralen Steuerung. Mit dem Befehl erhält die Lagersteuerung die Palettennummer, sowie Angaben über die Smartiebelegung auf der Matrix. Die Matrixbeschreibung der Lagerpalette sieht folgendermaßen aus: rgybw-*91. rgybw- beschreibt die Farben (r) red, (g) green, (y) yellow, (b) blue und (w) brown. Der Multiplikator sagt aus, dass die Palette 91 Smar-ties im Format 7x13 fasst. Diese Daten werden später mit in die Bestandsführung aufgenommen. Zunächst muss jedoch über die Bestandsdatei ein freies Fach innerhalb des Lagers ermittelt werden. Anschließend wird das Lager von der Lagersteuerung angewiesen, die Lagerpalette vom Transportband zu nehmen und in das ermittelte Lagerfach zu schieben.

2.3.5.1 Einlagerungsvorgang erfolgreich

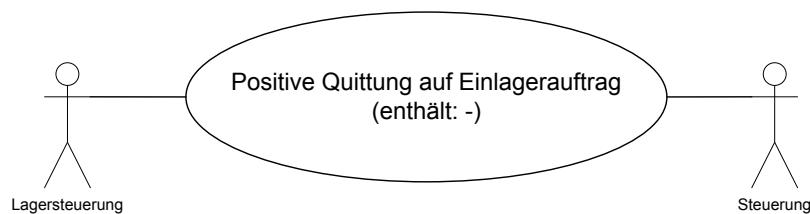


Abbildung 2.43: Einlagerauftrag von Steuerung positiv quittiert

Voraussetzung für einen erfolgreichen Einlagerungsvorgang ist die positive Quittung des Lagers. Durch die Quittung wird bestätigt, dass sich die Lagerpalette nun physikalisch im Lager befindet. Um den Einlagerungsvorgang abzuschließen muss die Lagersteuerung die Bestandsdaten aktualisieren. Anschließend erhält die Steuerung eine positive Quittung.

2.3 Das Lagersystem

2.3.5.2 Einlagerungsvorgang nicht erfolgreich

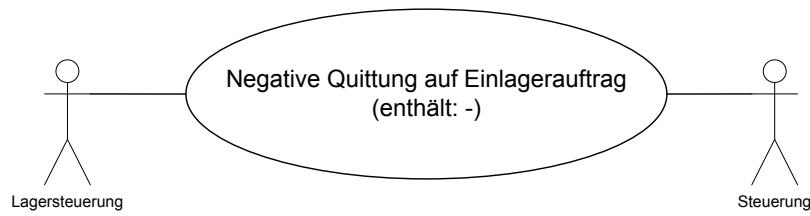


Abbildung 2.44: Einlagerauftrag von Steuerung negativ quittiert

Die Einlagerung kann aus folgenden Gründen fehlschlagen:

- kein freier Lagerplatz vorhanden
- einzulagernde Lagerpalette ist leer
- mechanischer Fehler (negative Quittung von Lager)

Tritt einer der ersten beiden Fehler auf, erhält die Steuerung sofort eine negative Quittung. Bei einem mechanischen Fehler wird zunächst der Operator hinzugezogen (??).

2.3 Das Lagersystem

2.3.5.6 Auslagern der Lagerpaletten

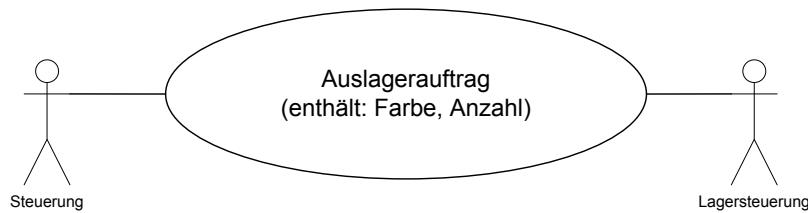


Abbildung 2.45: Auslagerauftrag von Steuerung an Lagersteuerung

Während der Abarbeitung eines Auftrags müssen dem Roboter die mit Smarties bestückten Lagerpaletten durch das Transportsystems zugeführt werden. Das zentrale Steuersystem weist die Lagersteuerung deshalb an, die entsprechenden Lagerpaletten auszulagern. Dabei erhält die Lagersteuerung die Farbe der von der Bestückungsstation benötigten Smarties, sowie deren Anzahl. Damit eine Lagerpalette mit den Smarties der angeforderten Farbe ausgelagert werden kann, muss zunächst herausgefunden werden, in welchem Fach sich die entsprechenden Smarties befinden. Dazu werden von der Lagersteuerung die intern gehaltenen Informationen über den Inhalt der Bestandsdatei ausgewertet. Die Auswahl der Lagerpalette erfolgt aufgrund zweier Regeln. Falls möglich wird eine angebrochene Palette ausgewählt, die einerseits genügend Smarties enthält, um den Auftrag zu erfüllen, andererseits aber den kleinsten Rest auf der Palette nach Erfüllen des Auftrags aufweist. Existiert keine nicht mehr vollständig bestückte Palette, welche die geforderte Menge erfüllen kann, wird die Palette mit dem ältesten Einlagerdatum gewählt. Sollte diese Palette nicht die ausreichende Menge Smarties beinhalten, um den Auftrag zu erfüllen, erkennt dies die Steuerung und leert diese restlos. Anschließend erteilt die Steuerung erneut einen Auftrag zur Auslagerung der besagten Smartiefarbe, diesmal jedoch mit der Aufforderung die benötigte Restmenge bereitzustellen. Die Steuerung wiederholt diesen Vorgang solange, bis die gewünschte Anzahl ausgelagert wurde.

2.3 Das Lagersystem

2.3.5.6.1 Auslagerungsvorgang erfolgreich

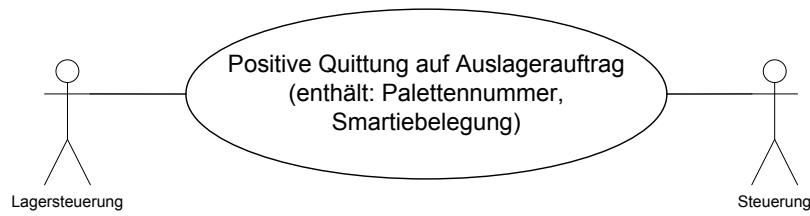


Abbildung 2.46: Auslagerauftrag von Steuerung positiv quittiert

Wurde die Lagerpalette aus dem Lagerschacht geholt und auf den Schlitten des Transportsystems geschoben, wird dies durch eine positive Quittung des Palettenlagers an die Lagersteuerung signalisiert. Die aus dem Lager genommene Lagerpalette wird nun aus dem Bestandsregister gelöscht. Daraufhin erfolgt die Bestätigung des Auslagerungsvorgangs an die Steuerung. Die Bestätigung beinhaltet gleichzeitig die Palettennummer, sowie die Smartiebelegung der ausgelagerten Lagerpalette. Mit Hilfe der Palettennummer kann die Lagerpalette von der Steuerung jederzeit eindeutig identifiziert werden, solange sie sich außerhalb des Lagers befindet. Die Smartiebelegung dient dem Roboter zur Koordination des Bestückungsvorgangs.

2.3 Das Lagersystem

2.3.5.6.2 Auslagerungsvorgang nicht erfolgreich

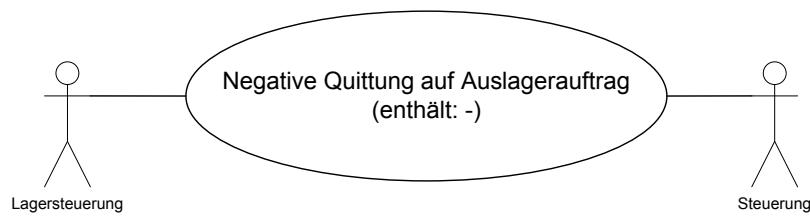


Abbildung 2.47: Auslagerauftrag von Steuerung negativ quittiert

Liegt z.B. ein mechanischer Defekt des Palettenlagers vor, so wird u.U. der Lageroperator hinzugezogen und gegebenenfalls der Befehl des Auslagerns negativ quittiert (??).

2.3 Das Lagersystem

2.3.5.7 Auftrag zum Herunterfahren des Lagersystems



Abbildung 2.48: Auftrag zum Herunterfahren des Lagersystems von der Steuerung

Am regulären Ende einer Serie von Aufträgen, oder als Notfallmaßname während einer Auftragsbearbeitung, kann die Steuerung den Auftrag zum kontrollierten Runterfahren des Systems geben. Nach Erhalt des Auftrags löst das Lagersystem kontrolliert die Verbindung zur Steuerung und zum Lager. Der Datenbestand, sowie der Auftragsstatus müssen an dieser Stelle nicht explizit gesichert werden, da er nach Ausführung jedes Auftrags sofort gesichert wird. Als letztes wird die Lagersteuerung beendet. Das Lager bleibt solange an, ist aber inaktiv. Es kann nach Herunterfahren der Lagersteuerung einfach ausgeschaltet werden.

2.3.6 Bestandsverwaltung

In diesem Abschnitt wird der Aufbau der Datenbasis als Datei erläutert.

2.3.6.1 Inhalt der Datei zur Bestandsverwaltung

Die momentane Beschaffenheit der FDZ erfordert aufgrund der Überschaubarkeit des Lagersystems keine Datenbank für die Bestandsführung der Smarties. Vielmehr werden folgende Informationen bezüglich des Lagerbestands in einer Datei abgespeichert:

- Name des Palettenlagers
- Absolute Anzahl aller Paletten-Stellplätze
- Anzahl der freien Paletten-Stellplätze

2.3 Das Lagersystem

- Anzahl der eingelagerten Smarties für jede Farbe
- Informationen zu eingelagerten Paletten
 - Palettennummer
 - Farbe der Smarties
 - Belegungsmatrix
 - Einlagerdatum

2.3.6.2 Aufbau der Datei

Die Bestandsdatei, `LAGERBESTAND.txt`, mit Hilfe derer die logische Bestandsverwaltung des Lagersystems durchgeführt wird, hat folgenden Aufbau:

1	<Name des Palettenlagers>
2	<Absolute Anzahl aller Paletten-Stellplätze>
3	<Anzahl der freien Paletten-Stellplätze>
4	[<Farbe><Anzahl>:]*[<Farbe><Anzahl>]
5	<Fach-Nr>:<Paletten-Nr>:<Matrixbelegung>:<Einlagerdatum>] -
6	<Fach-Nr>:<Paletten-Nr>:<Matrixbelegung>:<Einlagerdatum>] -
.	.
.	.
19	<Fach-Nr>:<Paletten-Nr>:<Matrixbelegung>:<Einlagerdatum>] -

Aufbau der Bestandsdatei

Zeile 1: Sollten zu einem späteren Zeitpunkt weitere Palettenlager in das System eingebunden werden, muss jedes Lager eindeutig identifiziert werden können. Hierzu dient der Name des Palettenlagers.

Zeile 2: In der zweiten Zeile befindet sich eine Konstante, welche die Gesamtanzahl der im Lager vorhandenen Paletten-Stellplätze angibt.

Zeile 3: In der dritten Zeile befindet sich ein Zähler, der die Anzahl der freien Paletten-Stellplätze innerhalb des Palettenlagers angibt.

Zeile 4: In der vierten Zeile wird zu jeder Farbe die entsprechende Anzahl der sich im Lager befindlichen Smarties gespeichert. Die Informationen werden durch einen Doppelpunkt voneinander getrennt.

Zeilen 5 - 19: Hier befinden sich Informationen zum jeweiligen Palettenfach. Am Anfang jeder Zeile steht die Palettenfach-Nummer. Befindet sich im Palettenfach eine Lagerpalette, folgt die entsprechende Paletten-Nummer, sowie die Matrixbelegung und das Einlagerdatum. Die Matrixbelegung beschreibt die Farbbelegung der Slots auf der Palette in der Form rgybw*91. Befindet sich keine Lagerpalette im Lagerschacht, steht an Stelle der Palettendaten ein Minus-Zeichen. Alle Informationen in einer Zeile werden durch einen Doppelpunkt getrennt.

2.3 Das Lagersystem

2.3.6.3 Beschreibungen der Nicht-Terminalsymbole

<Name des Palettenlagers> : Der Name kann aus einer beliebig langen Zeichenkette bestehen.

<Anzahl der freien Paletten-Stellplätze> : Zahl, welche die Anzahl der freien Paletten- Stellplätze angibt.

<Farbe> : Farbe der Smarties. Erlaubte Werte: r (rot), g (grün), b (blau), y (gelb), w (braun).

<Anzahl> : Anzahl der Smarties einer bestimmten Farbe.

<Fach-Nr> : Für die Palettenfach-Nummer sind Werte zwischen 0 und 999 erlaubt.

<Paletten-Nr> : Dreistellige Zeichenkette, welche jede Palette eindeutig identifiziert.

<Matrixbelegung> : Enthält Information über freie und belegte Felder auf der Lagerpalette. Die Matrixbelegung ist 91 Zeichen lang.

<Einlagerdatum> : Das Datum der Einlagerung wird in der Form DD.MM.JJJJ abgespeichert.

2.4 Das Transportsystem

2.4 Das Transportsystem

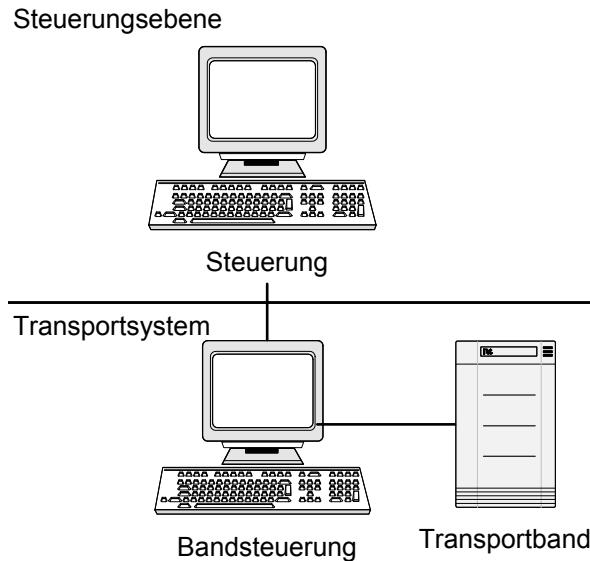


Abbildung 2.49: Aufbau des Transportsystems

Die Transportsteuerung erstellt aus den Befehlen der Steuerung eine Befehlskette, die das Transportband abarbeitet. Die Transportsteuerung führt auch eine Log-Datei für Recovery-Zwecke, wie in Abschnitt ?? beschrieben.

2.4.1 Aufbau des Transportbandes

Für einen besseren Überblick wird hier beschrieben, wie die unterste Ebene des Subsystems - das Transportband - aufgebaut ist, und welche Objekte hier eine Rolle spielen. Zunächst eine schematische Abbildung des Transportbandes:

2.4 Das Transportsystem

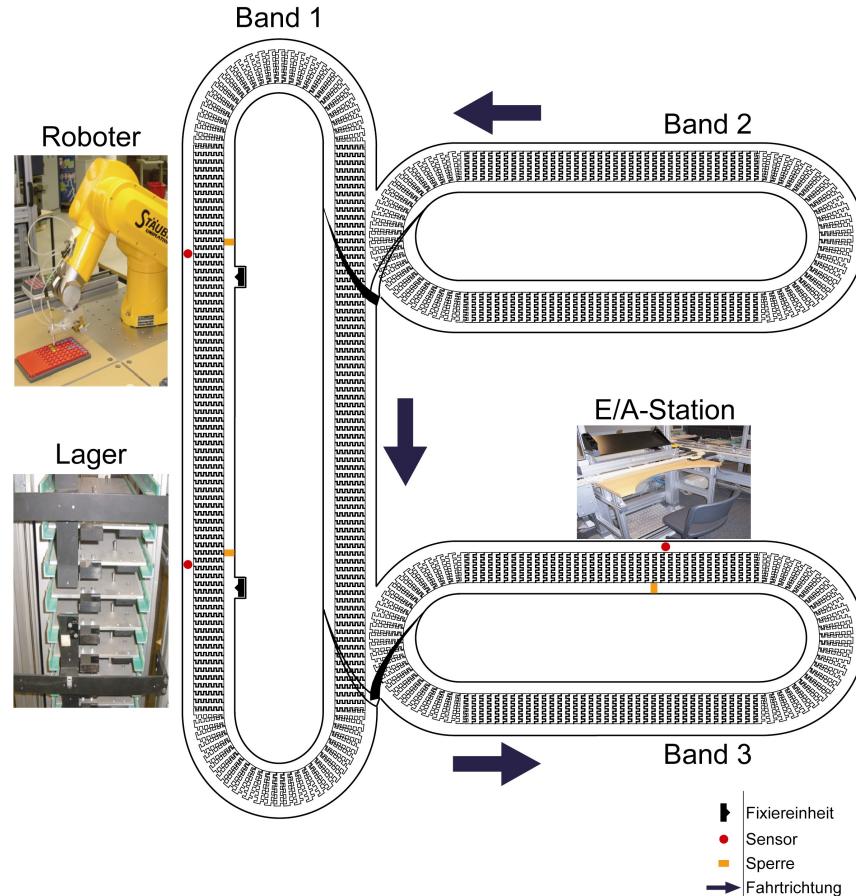


Abbildung 2.50: Physikalischer Aufbau des Transportsystems

2.4.2 Stationen und Fixierstationen

Es gibt drei Stationen auf dem Transportband, an die ein Schlitten transportiert werden muss: Zum Roboter, zum Lager und zur E/A-Station. Die einzelnen Positionen der Stationen an den Bändern, d.h. an welchen Sperren bzw. Sensoren sie sich befinden, wird in einer Bestandsdatei gespeichert. Die Stationen Roboter, Lager und E/A verfügen zusätzlich zu den Sperren noch über einen weiteren Mechanismus zum Fixieren der Schlitten: Eine Fixierstation. . Um einen reibungslosen Transport sicherzustellen darf sich immer nur ein Schlitten in einer der Kurven an den Bandenden oder im Bereich der Weichen befinden. Um das sicherzustellen befinden sich am Eingang der Kurven und Weichen Stopper, die durch das Transportband beim Einfahren eines Schlittens in die Kurve/Weiche hochgefahren und beim Ausfahren wieder runtergefahren werden müssen.

2.4 Das Transportsystem

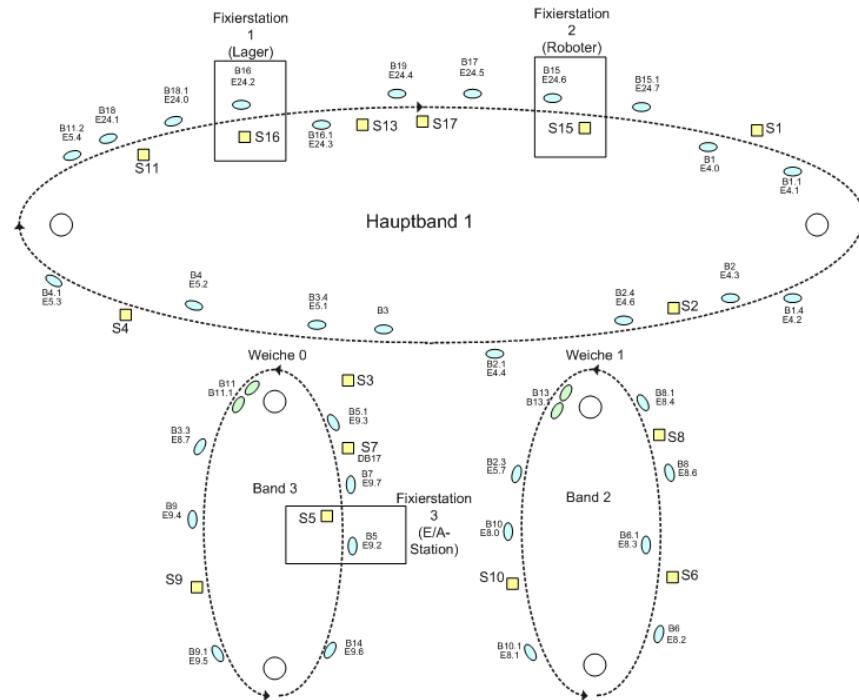


Abbildung 2.51: Schema des Transportsystems mit Sensoren und Stopfern (Ein- und Ausgänge)

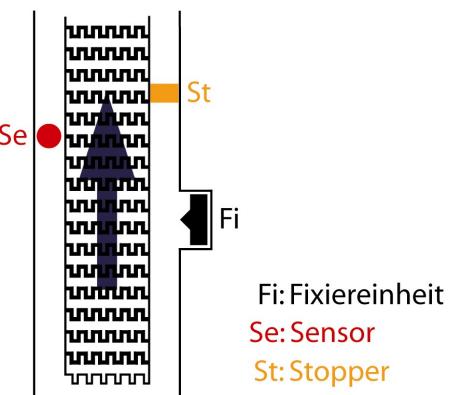


Abbildung 2.52: Anordnung von Sensor, Stopper und Fixiereinheit

2.4.3 Die Bänder

Das Transportband besteht aus drei Bändern: Dem großen Hauptband(Band 1) und zwei kleineren Bändern: Nebenband (Band 2) und E/A-Band (Band 3). Die Bänder sind miteinander verbunden. Die Bewegungsrichtung ist in der Abbildung eingezeichnet, sie kann nicht verändert werden. Nebenband 2 dient als 'Abstellplatz' und als Einsetzort für leere Schlitten.Nebenband 3 führt zur EA-Station. Schlitten die auf diesem Band fahren, bleiben solange auf diesem Nebenband bis ein Befehl zum runterleiten kommt.

2.4.4 Schlitten ID

Jedem auf dem Transportband befindlichen Schlitten wird eine eindeutige Ganzzahl zwischen 0 und der maximalen Schlittenanzahl-1 zugeordnet, wobei im Allgemeinen alle Schlitten von 0 beginnend durchnummieriert werden. Die ID wird von der SPS zugeordnet.

1. **Neue Schlitten ID Generieren** Im Allgemeinen wird immer dann eine neue ID vom Programm erzeugt, sobald ein Schlitten an einem Sensor ankommt, der keinen Schlitten erwartet! Die Nummern werden aus einem Nummernpool generiert, aus dem jeweils die kleinste freie Nummer ausgewählt wird. Eine neue ID kann praktisch an jedem Sensor erzeugt werden. Von der Nummerierung ausgeschlossene Sensoren:

- B16 (Lager)
- B10 (Nebenband 2, löscht nur Nummern, generiert keine neuen ID's!!!)
- B10.1
- B6
- B6.1
- B8

Theoretisch ist ein Einsetzen von Schlitten überall möglich, solange sich kein Schlitten in einem Bereich zwischen zwei Sensoren befindet (siehe Abbildung: ??) und der eingesetzte Schlitten nicht zu nah an dem bereits auf dem Transportband fahrenden Schlitten ist (siehe Abbildung: ??). Um jedoch einen Fehlerfreien Ablauf zu gewährleisten, darf ein neuer Schlitten nur auf Kommando von der Bandsteuerung (Schlitten ID -1) vom Nebenband 2 angefordert werden. Die neue Nummer wird dabei dann bei Sensor B8.1 generiert.

2. **Schlitten ID Weitergabe** Wenn ein Schlitten einen Sensor verlässt (Fallende Flanke), wird die ID an den nächsten Sensor übergeben (sog. 'Nachfolger'). An den Weichen wird die ID an zwei Nachfolger übergeben (auf dem Hauptband und entsprechenden Nebenband). Von der Weitergabe ausgeschlossene Sensoren:

- B16 (Lager)
- B10 (Nebenband 2, löscht nur Nummern, generiert keine neuen ID's!!!)
- B10.1
- B6

2.4 Das Transportsystem

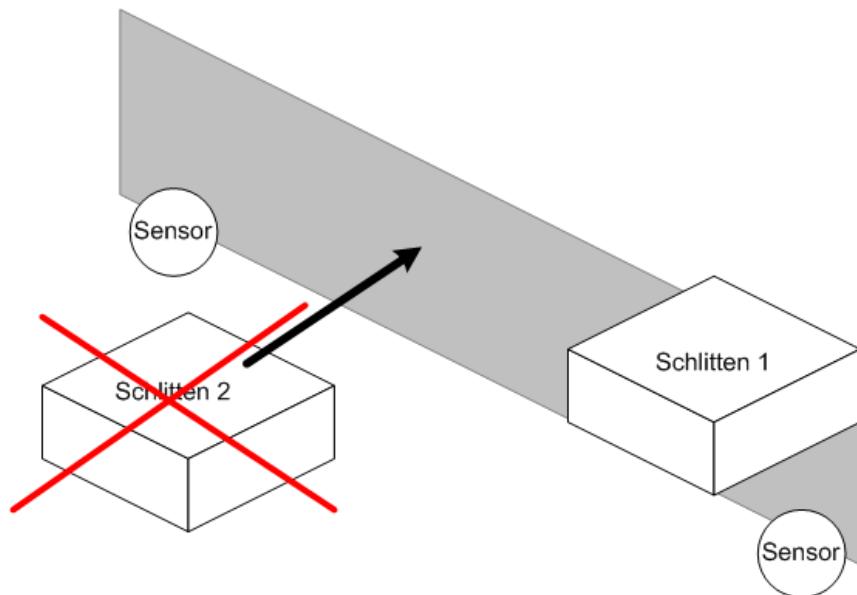


Abbildung 2.53: Falsches Einsetzen von Schlitten, zwei Schlitten in einem Bereich

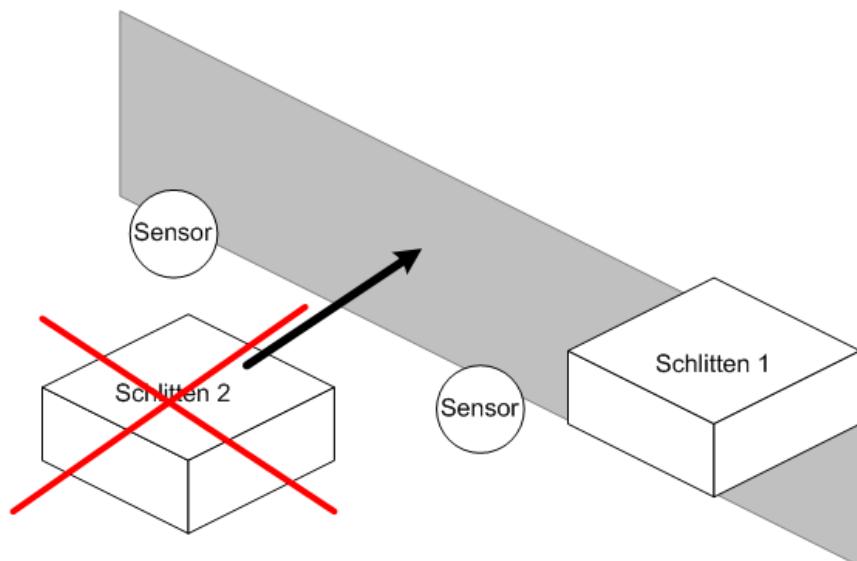


Abbildung 2.54: Falsches Einsetzen von Schlitten, Abstand der Schlitten zu klein

2.4 Das Transportsystem

- B6.1
 - B8
3. **Schlitten ID Löschen** Das Löschen der Schlitten ID ist nur an Sensor B10 möglich. Das bedeutet, dass der zu löschenende Schlitten auf Nebenband 2 ausgeschleust werden muss um ihn zu entnehmen.

2.4.5 Kommunikation

Die SPS soll in der Lage sein Befehle von ZenOn zu empfangen und sie sinnvoll zu verarbeiten. Folgende Befehle können verarbeitet werden:

- Anforderung eines freien Schlittens
- Abfrage Weichenstellung
- Abfrage Fixierstation
- Transportsystem herunterfahren
- Internes Kommando für Rücksetzung von Ack. 2
- FehlerId- Weitergabe

2.4.6 Die Weichen

Um zu steuern, ob ein Schlitten in eines der Nebenbänder einfahren soll, oder auf dem Hauptband verbleibt, gibt es je Nebenband eine pneumatisch gesteuerte Weiche.

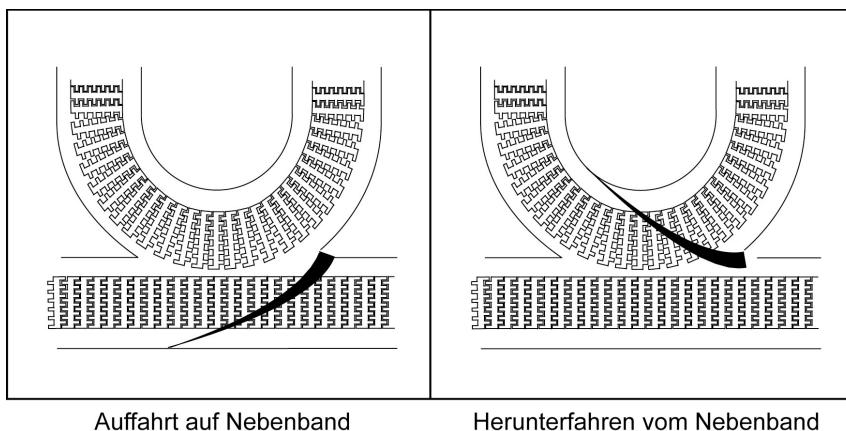


Abbildung 2.55: Stellungen der Weichen

Die Weichen können zwei Stellungen haben, die auch in der Abbildung eingezeichnet sind:

- In einer Stellung befindet sich die Weiche auf dem Hauptband und bewirkt, dass Schlitten in das Nebenband einfahren.

2.4 Das Transportsystem

- In der anderen Stellung werden Schlitten, die sich auf einem Nebenband befinden, auf das Hauptband zurückgeschleust.

Eine Weiche wird über zwei binäre Ausgänge gesteuert. Ein Ausgang lässt die Weiche auf der andere lässt sie zufahren. Zusätzlich befinden sich vor jeder der Weichen Stopper mit den entsprechenden Sensoren, die dafür sorgen, dass nur Teile von einer der beiden „Zufahrten“ in den Wirkbereich der Weiche fahren können. An den beiden Ausgängen verfügen die Weichen ausserdem über Sensoren, die melden, wenn der Schlitten den Bereich der Weiche verlassen hat. Beide Weichen befinden sich bei Systemstart in Stellung 0, d.h. wie im Abbildung ?? beim Herunterfahren vom Nebenband.

Es werden zwei Arten von Weichen unterschieden:

- Weiche 0

Weiche 0 befindet sich zwischen Hauptband und Nebenband 3.

- Weiche1

Weiche 1 befindet sich zwischen Hauptband und Nebenband 2.

2.4.7 Die Sperren

Die Sperren dienen dazu, Schlitten zu stoppen (beispielsweise an einer der drei Stationen). Damit ist sichergestellt, dass der Schlitten so lange an einer bestimmten Stelle verbleibt, wie er dort benötigt wird, auch wenn sich das Transportband während seines Aufenthaltes weiter bewegt. Es befinden sich an jeder Station, wie auch an verschiedenen Stellen und Kurven der Bänder Sperren. Die Sperren können so auch zur Realisierung einer effektiven Wegplanung verwendet werden. Es werden zwei Arten von Sperren unterschieden:

Stopper Ein Stopper besteht im wesentlichen aus mindestens einem pneumatischen Aktor, der einen Träger auf dem Band anhalten kann und zwei Sensoren. Der eine Sensor meldet, wenn ein Teil vom Aktor angehalten wird. Der zweite Sensor ist nach dem Stopper angebracht und meldet, wenn ein Schlitten den Stopper passiert hat.

Fixierstation Eine Fixierstation besteht aus zwei Stoppern und 3 Sensoren (Abbildung ??). Der erste Stopper S1 dient im wesentlichen nur dazu, Schlitten vor der vollen Station zu stauen. Sobald ein einzelner Schlitten diesen Stopper passiert hat, wird er durch den zweiten Stopper angehalten und fixiert.

Ausgangssituation:

- Stopper S1 eingefahren
- Stopper S2 eingefahren

Fixieren:

- Sensor B2 fährt Stopper S2 raus
- Sensor B3 löst Fixiervorgang
- Wenn ein weiterer Schlitten den Sensor B1 überfährt wird Stopper S1 ausgefahren

2.4 Das Transportsystem

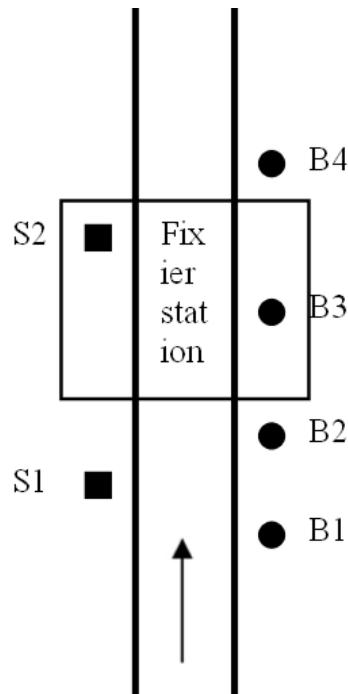


Abbildung 2.56: Schema einer Fixierstation (Sx: Stopper, Bx: Digitaler Sensor)

Freigeben:

- Stopper S2 runterfahren
- Fixierung aufgehoben
- Wenn Sensor B4 überfahren wird, wird Stopper S1 eingefahren

2.4.7.1 Die Sensoren

Die Sensoren geben Auskunft darüber, an welcher Stelle auf dem Transportband sich gerade Schlitten befinden. Sie geben ein Signal aus, wenn ein Schlitten den Sensor passiert.

Auf dem Transportband befinden sich an vielen Stellen Sensoren, insbesondere vor jeder Sperre. Damit kann das System überhaupt erst erkennen, wann ein Schlitten an einer der Stationen angekommen ist und die Sperren aktiviert werden müssen, um den Schlitten zu fixieren.

Belegung der Ein- und Ausgänge

Symbol	Adresse	Typ	Kommentar
Ausgänge			
Band3_Motor	A 12.0	BOOL	Antrieb Band 3 Ein
Band2_Motor	A 12.1	BOOL	Antrieb Band 2 Ein
Band1_Motor	A 12.2	BOOL	Antrieb Band 1 Ein
Lampe Störung 17H3	A 13.0	BOOL	Lampe Störung 17H3

2.4 Das Transportsystem

Symbol	Adresse	Typ	Kommentar
Lampe Not-Aus 17H5	A 13.1	BOOL	Lampe Not-Aus 17H5
Lampe Steuerung Ein 17H7	A 13.2	BOOL	Lampe Steuerung Ein 17H7
Lampe Start Aktiv 17H9	A 13.3	BOOL	Lampe Start Aktiv 17H9
Stopper 1 vor	A 16.0	BOOL	Stopper 1 vor
Stopper 1 zurück	A 16.1	BOOL	Stopper 1 zurück
Stopper 2 vor	A 16.2	BOOL	Stopper 2 vor
Stopper 2 zurück	A 16.3	BOOL	Stopper 2 zurück
Stopper 3 vor	A 16.4	BOOL	Stopper 3 vor
Stopper 3 zurück	A 16.5	BOOL	Stopper 3 zurück
Stopper 4 vor	A 16.6	BOOL	Stopper 4 vor
Stopper 4 zurück	A 16.7	BOOL	Stopper 4 zurück
Weiche2_auf	A 17.0	BOOL	Weiche 2 Auf W2
Weiche2_zu	A 17.1	BOOL	Weiche 2 Zu W2
Stopper 10 vor	A 17.2	BOOL	Stopper 10 vor ST10
Stopper 10 zurück	A 17.3	BOOL	Stopper 10 zurück ST10
Stopper 6 vor	A 17.4	BOOL	Stopper 6 vor ST6
Stopper 6 zurück	A 17.5	BOOL	Stopper 6 zurück ST6
Fix_0_an	A 17.6	BOOL	Ausgang=true_1 dann fährt Fix E/A aus
Stopper 8 vor	A 20.0	BOOL	Stopper 8 vor ST8
Stopper 8 zurück	A 20.1	BOOL	Stopper 8 zurück ST8
Weiche1_auf	A 21.0	BOOL	Weiche 1 Auf W1
Weiche1_zu	A 21.1	BOOL	Weiche 1 Zu W1
Stopper 5 vor	A 21.2	BOOL	Stopper 5 vor ST5
Stopper 5 zurück	A 21.3	BOOL	Stopper 5 zurück ST5
Stopper 9 vor	A 21.4	BOOL	Stopper 9 vor ST9
Stopper 9 zurück	A 21.5	BOOL	Stopper 9 zurück ST9
Stopper 7 vor	A 21.6	BOOL	Stopper 7 vor ST7
Stopper 7 zurück	A 21.7	BOOL	Stopper 7 zurück ST7
Stopper S15	A 24.0	BOOL	Stopper S15 true=einfahren
Stopper S17	A 24.1	BOOL	Stopper S17 true=einfahren
Fix_2_an	A 24.2	BOOL	Ausgang=true/1 dann fährt Fix Roboter aus
Stopper S13	A 24.3	BOOL	Stopper 13 nach Fixierstation Lager (true=eingefahren)
Stopper S16	A 24.4	BOOL	Stopper 16 Fixierstation Lager (true=einfahren)
Stopper S11	A 24.5	BOOL	Stopper 11 (true=rausfahren)

2.4 Das Transportsystem

Symbol	Adresse	Typ	Kommentar
Fix_1_an	A 24.6	BOOL	Wenn Ausgang auf true/1 dann fährt Fixierstation an Lager aus
Eingänge			
Not Aus betätigt	E 0.0	BOOL	Not Aus betätigt
Steuerung Ein	E 0.1	BOOL	Steuerung Ein
Starttaster	E 0.2	BOOL	Starttaster
Stoptaster	E 0.3	BOOL	Stoptaster
Reset	E 0.4	BOOL	Reset
Initiator B1	E 4.0	BOOL	Stopper 1 belegt
Initiator B1.1	E 4.1	BOOL	Stopper1 frei
Initiator B1.4	E 4.2	BOOL	Am Ende von Kurve 2
Initiator B2	E 4.3	BOOL	Stopper 2 belegt
Initiator B2.1	E 4.4	BOOL	Palette auf Band 1
Initiator B2.4	E 4.6	BOOL	Stopper2 frei
Initiator B3	E 4.7	BOOL	Stopper 3 belegt
Initiator B3.4	E 5.1	BOOL	Stopper3 frei
Initiator B4	E 5.2	BOOL	Stopper 4 belegt B4
Initiator B4.1	E 5.3	BOOL	Stopper4 frei
Initiator B11.2	E 5.4	BOOL	Am Ende von Kurve 1
Initiator B2.3	E 5.7	BOOL	Palette auf Band 2
Initiator B10	E 8.0	BOOL	Stopper 10 belegt
Initiator B10.1	E 8.1	BOOL	Stopper10 frei
Initiator B6	E 8.2	BOOL	Stopper 6 belegt
Initiator B6.1	E 8.3	BOOL	Stopper 6 frei
Initiator B8.1	E 8.4	BOOL	Stopper 8 frei
Initiator B8	E 8.6	BOOL	Stopper 8 belegt
Initiator B3.3	E 8.7	BOOL	Palette auf Band 3
Initiator B5	E 9.2	BOOL	Stopper 5 belegt
Initiator B5.1	E 9.3	BOOL	Stopper5 frei
Initiator B9	E 9.4	BOOL	Stopper 9 belegt
Initiator B9.1	E 9.5	BOOL	Stopper9 frei
Initiator B14	E 9.6	BOOL	Kurve Band 3
Initiator B7	E 9.7	BOOL	Stopper 7 belegt
Initiator B18.1	E 24.0	BOOL	Am Ende von Kurve 1
Initiator B18	E 24.1	BOOL	Stopper 11 belegt
Initiator B16	E 24.2	BOOL	Stopper 16 belegt
Initiator B16.1	E 24.3	BOOL	Stopper 16 frei
Initiator B19	E 24.4	BOOL	Stopper 13 frei
Initiator B17	E 24.5	BOOL	Stopper 17 frei
Initiator B15	E 24.6	BOOL	Stopper 15 belegt
Initiator B15.1	E 24.7	BOOL	Stopper 15 frei

2.4 Das Transportsystem

2.4.8 Wichtige Daten der Transportsteuerung

Die Transportsteuerung muss mindestens die folgenden Dinge "wissen":

- Welche der drei Stationen (Roboter, Lager, E/A-Station) sich bei welchem Sensor befindet.
- Wo sich welcher Schlitten zu einem bestimmten Zeitpunkt befindet.
- Welche Fixierstationen wie gestellt sind.
- In welchem Zustand sich das Transportsystem gerade befindet (siehe Abschnitt ??).

Die Anzahl der Schlitten wird vor dem Starten des Subsystems festgelegt, ebenso deren Startposition und die Weichen- und Sperrenstellung (siehe Abschnitt ??).

Anzahl der Schlitten

Eine wichtige Frage ist, wie viele Schlitten sich gleichzeitig auf dem Transportband befinden können.(In unserer Fall sind maximal 5 Schlitten vereinbart.) Die Anzahl der Schlitten sollte sinnvoll gewählt werden. Sie wird in einer Bestandsdatei gespeichert. Die festgelegte Anzahl von Schlitten muss sich vor dem Hochfahren des Systems auf dem Band befinden (siehe Initialzustand in Abschnitt ??).

Als Maximalzahl der Schlitten, die sich gleichzeitig auf dem Band befinden dürfen, kann man eine Zahl festlegen, die kleiner als die Zahl der Sperren auf dem Transportband ist.

2.4.8.1 Festlegungen und Rahmenbedingungen

Im Folgenden werden einige Festlegungen aufgeführt, die im weiteren Dokument als Voraussetzung gelten:

- Durch die Stopperlogik ist ein Auffahren der Schlitten ausgeschlossen.
- Nach jedem Hochfahren des Subsystems wird der Zustand beim Ausschalten wieder hergestellt.
- Auf Nebenband 3 darf nur ein Schlitten in umlauf sein.
- Während eines Arbeitsvorganges an einem Schlitten, werden die anderen Schlitten durch die Sensoren und Stopfern kontrolliert und eventuell gestoppt oder befreit.
- Die Anzahl der Schlitten auf dem Band bleibt während der Ausführung des aktuellen Bestückungsauftrages gleich. Neue Schlitten können nur auf dem Nebenband 2 eingesetzt bzw. entnommen werden, d.h. es können nur neue Schlitten auf das Band gelangen wenn auf dem Nebenband 2 sich Schlitten befinden und diese Angefordert werden.

2.4 Das Transportsystem

2.4.8.2 Timer

Um zu verhindern, dass eingeklemmte Schlitten vom Benutzer unbemerkt bleiben, wurde eine Timeout-Steuerung implementiert, die nach 10 Sekunden eine Fehlermeldung auslöst und das Band stoppt. Besonders gefährdete Stellen am Transportband sind dabei die Weiche, Fixierstationen und Kurven. Ist zwischen 2 Sensoren ein Stopper ausgefahren, wird dies natürlich am Timeout berücksichtigt! Die Fehlerbehandlung dazu ist im Benutzerhandbuch genau beschrieben.

2.4.8.3 Verkehrsregelung der Schlittens auf dem Band:

Es wurden bestimmte Verkehrsregeln festgelegt um einen reibungslosen Verkehr ohne Verklemmung oder Stau zu gewährleisten:

- Kurvenregelung: Wenn ein Schlitten in eine Kurve einfährt, dann wird der Stopper vor der Kurve hochgefahren damit es kein Stau in der Kurve gibt und kein Zusammenstoss zwischen den Schlitten verursacht wird. Somit befindet sich immer nur genau ein Schlitten in einer Kurve.
- An den Stationen: Wenn ein Schlitten sich an einer Station befindet, dann werden die Stopper vor der entsprechenden Station hochgefahren, um Zusammenstöße zwischen den Schlitten zu vermeiden.
- Die Schlitten bleiben immer auf dem Band auf dem sie sich gerade befinden, bis ein anderslautendes Kommando einer übergeordneten Instanz empfangen wird
- Weichenregelung: Der Schlitten der zuerst den Weichenraum erreicht, passiert diesen auch zuerst. Der Schlitten auf dem anderem Band wird solange vom Stopper angehalten.

2.4.8.4 Bestandsverwaltung

Damit die Bandsteuerung befähigt ist eine Wegplanung durchführen zu können, muss diese zu jedem Zeitpunkt über die Positionen der belegten Schlitten informiert sein. Diese Information entsteht indem ein leerer Schlitten angefordert bzw. ein belegter Schlitten bewegt wird. Ein Schlitten gilt solange als belegt bis er wieder freigegeben wird.

Die besagten Daten werden in der Bestandsdatei `transport.properties` von der Transportsteuerung wie folgt abgelegt:

1	Schlitten<SchlittenId>=<Band><Fixierstation><Belegungsstatus>
...	...
6	controllIP=<IP-Adresse>
7	Ziel=<Ziel>

2.4.8.4.1 Aufbau der Bestandsdatei

Aufbau der Bestandsdatei

2.4.8.4.2 Beschreibung der Nicht-Terminalsymbole

Beschreibung der Nicht-Terminalsymbole

<Band>: Identifikationsnummer des Bands; Zahl zwischen 0 und 9.
<Fixierstation>: Identifikationsnummer der Fixierstation; Zahl zwischen 0 und 999.
<IP-Adresse>: IP-Adresse der Steuerung; x.y.y.y; x = Zahl zwischen 1 und 223; y = Zahl zwischen 0 und 255.
<Ziel>: Ziel des angeforderten Schlittens; mögliche Werte: ro, la, ea.
<SchlittenId>: Identifikationsnummer des Schlittens; Zahl zwischen 0 und 99.
<Belegungsstatus>: Information über Inhalt des Schlittens; 0 für leer, 1 für bestückt.

2.4.9 Zustände

In diesem Abschnitt wird beschrieben, in welchen Zuständen sich die Transportsteuerung befinden kann. Alle Bänder bleiben in Bewegung in allen Zuständen ausser dem Fehlerzustand.

Es werden die folgenden Zustände des Transportsystems unterschieden:

- Aus-Zustand / Initialzustand
- Arbeitender Zustand
- Fehlerzustand
- Wartezustand

Das Band bleibt nur während des Fehlerzustandes stehen. Am Ende des Abschnittes wird noch darauf eingegangen, wie diese Zustände ineinander übergehen.

2.4.9.1 Aus-Zustand / Initialzustand

Dieser Zustand ist wie folgt definiert:

- N Schlitten befinden sich auf dem Band (N muss sinnvoll gewählt werden, siehe Abschnitt ??)
- Diese Schlitten sind unbeladen
- Die Sperren sind inaktiv

Mögliche Folgezustände:

- Wartezustand
- Fehlerzustand

2.4 Das Transportsystem

2.4.9.2 Arbeitender Zustand

Im arbeitenden Zustand werden Schlitten positioniert, d.h. Kommandos von Steuerung werden ausgeführt.

Mögliche Folgezustände:

- Wartezustand
- Fehlerzustand

2.4.9.3 Wartezustand

Alle Kommandos wurden erfolgreich abgearbeitet, d.h. beispielsweise alle Schlitten sind positioniert. Transportsteuerung ist bereit, Kommandos von Steuerung entgegenzunehmen und auszuführen oder das Subsystem kontrolliert herunterzufahren.

Mögliche Folgezustände:

- Arbeitender Zustand
- Fehlerzustand
- Aus-Zustand / Initialzustand

2.4.9.4 Fehlerzustand

Der Fehlerzustand tritt ein, wenn ein Fehler auftritt bzw. wenn ein Recovery notwendig ist. Siehe auch den Abschnitt ?? zur Fehlerbehandlung.

Mögliche Folgezustände:

- Arbeitender Zustand
- Wartezustand
- Aus-Zustand

2.4 Das Transportsystem

2.4.9.5 Zustandsübergänge

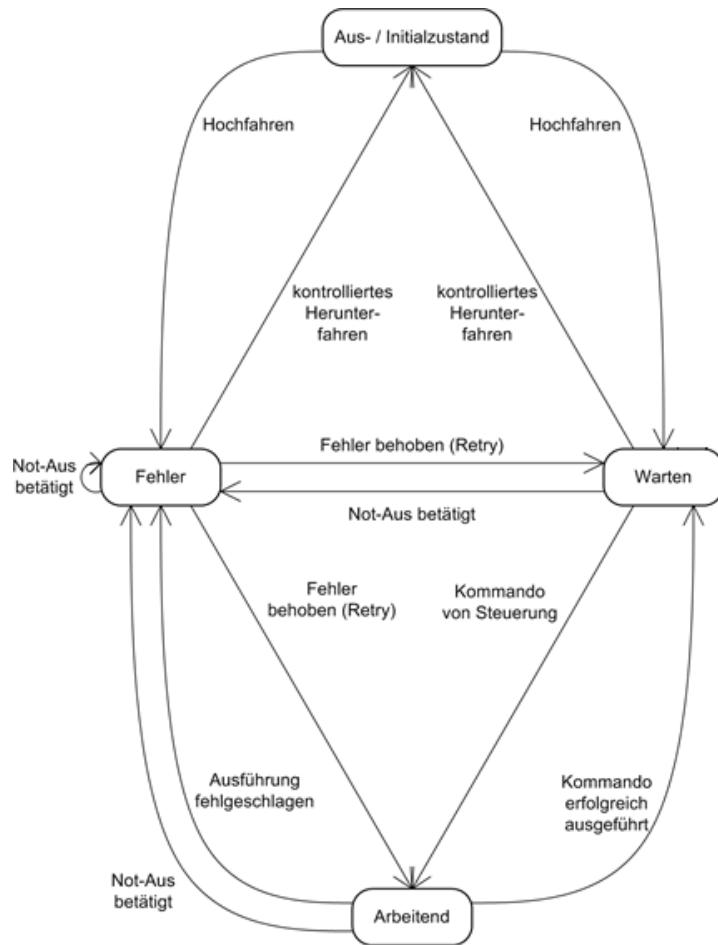


Abbildung 2.57: Zustandsübergänge

Hier werden die Übergänge zwischen den Zuständen beschrieben.

Hochfahren

Nachdem das Transportsystem aktiviert wurde, wird es hochgefahren. Zuerst registriert sich Zeno bei der Transportsteuerung, dann die Transportsteuerung bei der Steuerung.

Wenn die vorangegangene Aktion erfolgreich abgeschlossen wurde, wird die temporäre Log-Datei überprüft. Ist diese nicht leer, dann muss ein Recovery durchgeführt werden, d.h. das Subsystem wechselt in den Fehlerzustand.

Tritt während des Hochfahrens kein Fehlerzustand ein, geht das Subsystem in den Wartezustand über.

Kontrolliertes Herunterfahren

2.4 Das Transportsystem

Das Transportsystem kann aus dem Wartezustand oder dem Fehlerzustand heruntergefahren werden. Beim kontrollierten Herunterfahren werden alle Hardware-Komponenten abgeschaltet.

Kommando von Steuerung

Sobald die Steuerung ein Kommando schickt, das von der Transportsteuerung ausgeführt werden kann, geht das Transportsystem in den arbeitenden Zustand über, wenn es sich vorher im Wartezustand befunden hat.

Kommando erfolgreich ausgeführt

Wenn die Transportsteuerung ein Kommando erfolgreich abgearbeitet hat, wechselt das Transportsystem wieder in den Wartezustand zurück. Das Band bleibt in Bewegung.

Ausführung fehlgeschlagen

Trat bei der Ausführung eines Kommandos von Steuerung ein Fehler auf, geht das Transportsystem in den Fehlerzustand über.

Fehler behoben (Wiederholung)

Befindet sich das Transportsystem im Fehlerzustand, wird für den Operator eine Maske angezeigt, in der er das Wiederholen des fehlgeschlagenen Vorgangs auslösen kann. Wenn der Operator die Retry-Aktion auslöst, gelangt das Subsystem zunächst wieder in den Zustand, in dem es sich vor dem Fehlerzustand befand. Es kann also entweder in den arbeitenden oder in den Wartezustand übergehen. Wenn beispielsweise im Wartezustand eine Sperre ausfällt, erhält Bandsteuerung vom Transportband eine entsprechende Benachrichtigung. Daraufhin geht das Transportsystem in den Fehlerzustand über, und die bereits erwähnte Maske wird angezeigt. Der Operator kann das Problem mit der Sperre dann manuell beheben und einen Retry auslösen. Dies bewirkt, dass das Transportsystem wieder in den Wartezustand übergeht. Im arbeitenden Zustand ist der Ablauf ähnlich: Wenn das Transportsystem gerade einen Schlitten positioniert und z. B. das Band ausfällt, wechselt das Subsystem in den Fehlerzustand. Der Operator hat nun wieder die Möglichkeit, das Problem zu beheben und den Retry auszulösen. Bei einem Retry wird das Subsystem sofort wieder in den arbeitenden Zustand übergehen.

2.4.10 Steuerung an Bandsteuerung

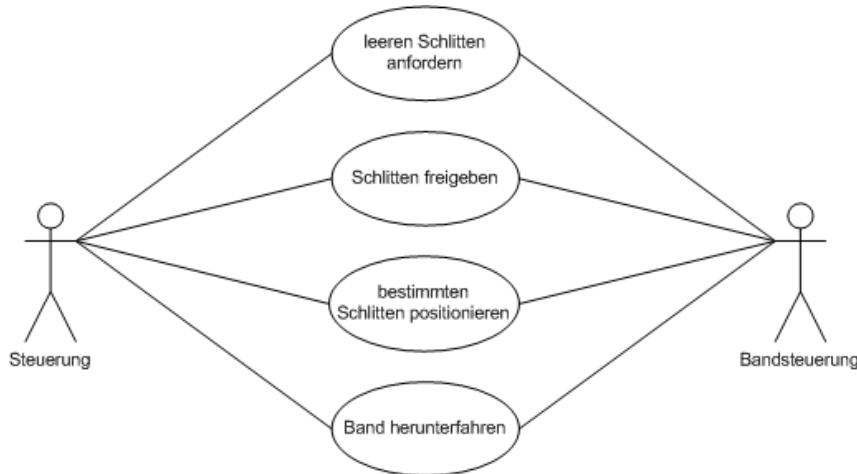


Abbildung 2.58: Steuerung an Bandsteuerung

Die Kommunikation zwischen Steuerung und Bandsteuerung bezieht sich größtenteils auf Tätigkeiten, die im Zusammenhang mit den Schlitten stehen.

Zunächst besteht für die Steuerung die Möglichkeit, einen leeren Schlitten anzufordern. Dabei wird der Bandsteuerung nur mitgeteilt, an welcher Stelle dieser benötigt wird. Welcher leere Schlitten benutzt wird und auf welchem Weg dieser an die gewünschte Position gelangt, ist der SPS überlassen. Sobald der Schlitten angefordert wurde, ist er reserviert, und der Steuerung wird seine ID mitgeteilt. Wenn ein Schlitten reserviert ist, kann er nicht mehr als leerer Schlitten angefordert werden. Ein reservierter Schlitten kann umpositioniert und freigegeben werden. Das Freigeben ist deshalb so wichtig, da der Schlitten automatisch auf Band 2 ausgeschleust wird und für andere Aufgaben zur Verfügung gestellt werden kann.

Ein ähnliches Prinzip wird bei der Positionierung eines schon mit einer Palette beladenen Schlittens benutzt. Ein solcher Schlitten muss natürlich vorher reserviert worden sein. Hierbei wird der Transportsteuerung dessen ID und der Ort, an den er gefahren werden soll, mitgeteilt. Anders wie bei der Bereitstellung eines leeren Schlittens kümmert sich die Transportsteuerung um die Wegplanung. Nachdem der positionierte Schlitten von der Steuerung nicht mehr benötigt wird, muss die Transportsteuerung darüber informiert werden, der Schlitten muss also freigegeben werden. Alternativ kann Steuerung auch befehlen, ihn an eine andere Position zu fahren.

Im Falle einer Störung, oder falls der Auftrag abgearbeitet worden ist, besteht für die Steuerung die Möglichkeit, die Transportsteuerung zu benachrichtigen und ihr das Herunterfahren des Transportsystems zu befehlen. Erhält die Transportsteuerung diese Instruktion, ist es ihre Aufgabe, das Transportsystem in einen konsistenten Zustand zu bringen, von dem ausgehend beim nächsten Hochfahren weitergearbeitet werden kann, und es dann abzuschalten. Konsistenter Zustand heißt im Wesentlichen nur, dass Hardware abgeschaltet wird. Der letzte Zustand ist immer in der `transport.properties` gespeichert. Wird am System etwas geändert, so muss diese Änderung manuell in die `transport.properties` eingetragen werden.

2.4 Das Transportsystem

2.4.11 Transportsteuerung an Transportband

Da das Transportband die unterste Ebene des Transportsystems darstellt, besteht die Kommunikation zwischen der Transportsteuerung und ihm darin, hardwareseitig in den Systemablauf einzugreifen. Dabei wird sich um das Stellen der Weichen, Sperren und Fixierstationen, die zur Positionierung der Schlitten benötigt werden, gekümmert.

2.4.12 Transportband an Transportsteuerung

Das Transportband ist dafür zuständig, Sensor-Signale durch die ZenOn-Schnittstelle, an die Transportsteuerung zu senden. Diese werden dann dazu verwendet, die einzelnen Schlitten zu überwachen und anhand dieses Wissens die Wegplanung zu realisieren.

Weiterhin teilt das Transportband der Transportsteuerung mit, dass bestimmte Fehler aufgetreten sind, so dass diese sich entweder selbst darum kümmert oder sie an die darüber liegende Steuerung weiterleitet.

2.4.13 Bandsteuerung an Steuerung

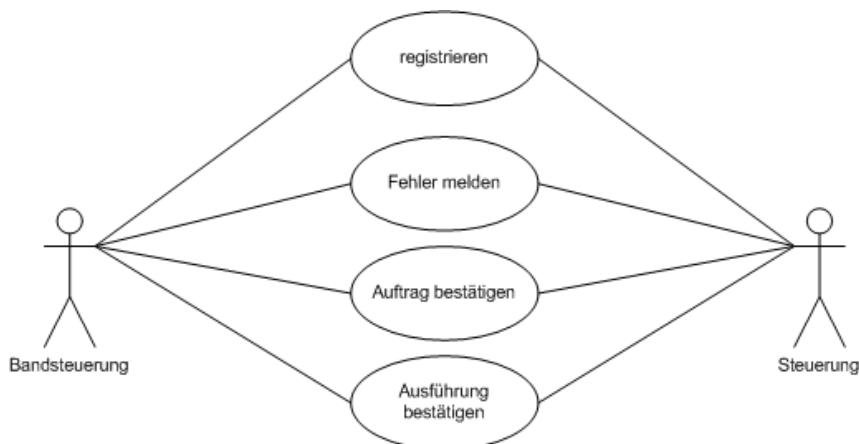


Abbildung 2.59: Bandsteuerung an Steuerung

Beim Hochfahren des Transportsystems ist es zunächst Aufgabe der Transportsteuerung, sich bei der Steuerung zu registrieren. Dadurch wird der Steuerung mitgeteilt, dass die Transportsteuerung nun vollständig einsatzbereit ist und mit ihr gearbeitet werden kann.

Treten während der durchgeführten Aufgaben in der Transportsteuerung nicht behandelbare Fehler auf - sei es, dass diese dort ihren Ursprung haben oder vom darunter liegenden Transportsystem gemeldet wurden - werden diese an die Steuerung weitergeleitet.

Die von Steuerung an Transportsteuerung gegebenen Aufträge werden nach deren Erhalt und nach erfolgreicher Ausführung durch eine Mitteilung an Steuerung bestätigt.

2.4 Das Transportsystem

2.4.14 Bandsteuerung

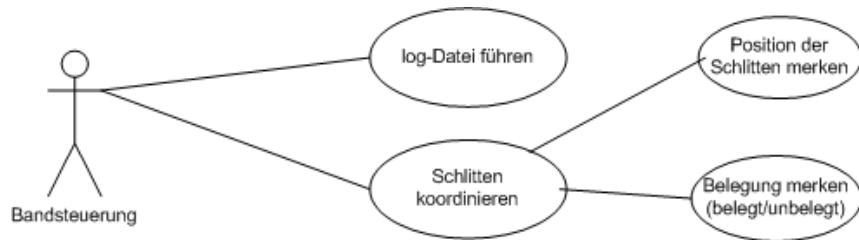


Abbildung 2.60: Bandsteuerung

Da die Transportsteuerung sich darum kümmert, wie ein Schlitten am besten von A nach B kommt, müssen diese irgendwie koordiniert werden. Dazu muss sie sich die Positionen der beladenen Schlitten merken. Die Transportsteuerung muss wissen, welcher Schlitten verwendet wird.

Des weiteren führt die Transportsteuerung Log-Dateien, mit deren Hilfe es möglich ist bei einem Wieder-Anlaufen des Systems eventuell die zuletzt durchgeführte Tätigkeit wieder aufzunehmen.

2.5 Die Eingabe/Ausgabe-Station (E/A-System)

2.5 Die Eingabe/Ausgabe-Station (E/A-System)

Derzeit ist die Eingabe/Ausgabe-Station wie folgt definiert:

- Bandsteuerung kennt einen Haltepunkt, welcher die Position des E/A-System's festlegt.
- Ein Mensch übernimmt sämtliche Aufgaben, die an der Eingabe/Ausgabe-Station erledigt werden müssen.
- Ein Schalter dient zur Bestätigung der Ausführung einer Aktion.
- Ein PC zeigt alle Anweisungen und die Ausführung von Aktionen an (inklusive der Maske im Fehlerzustand).
- Die gewünschte Anzahl der angeforderten Smarties beim Befüllen einer Lagerpalette kann immer bereitgestellt werden (unendlich großes Lager).

Das E/A-System besteht also aus der E/A-StationsSteuerung und einem "Helper". Der Helper erhält seine Arbeitsanweisungen über den PC und bestätigt deren Ausführung über den Schalter.

Auch das E/A-System führt eine Log-Datei für Recovery-Zwecke, wie in Abschnitt ?? beschrieben.

Um eine eindeutige Zuordnung der Paletten-IDs (Lagerpaletten) sicherstellen zu können ist eine Bestandsdatei notwendig. Außerdem kann dadurch die Bestückungsmatrix der einzuschleusenden Lagerpaletten zurückgegeben werden. Die Datei „eaQuantityfile.txt“ ist wie folgt aufgebaut:

1	<Name der E/A-Station>
2	<Absolute Anzahl der zugewiesen Paletten-IDs>
3	<Nächste freie Paletten-ID>
4	[<Paletten-ID>:<Matrixbelegung>]
5	[<Paletten-ID>:<Matrixbelegung>]
.	...
1002	[<Paletten-ID>:<Matrixbelegung>]

2.5 Die Eingabe/Ausgabe-Station (E/A-System)

2.5.1 Steuerung an E/A-StationsSteuerung

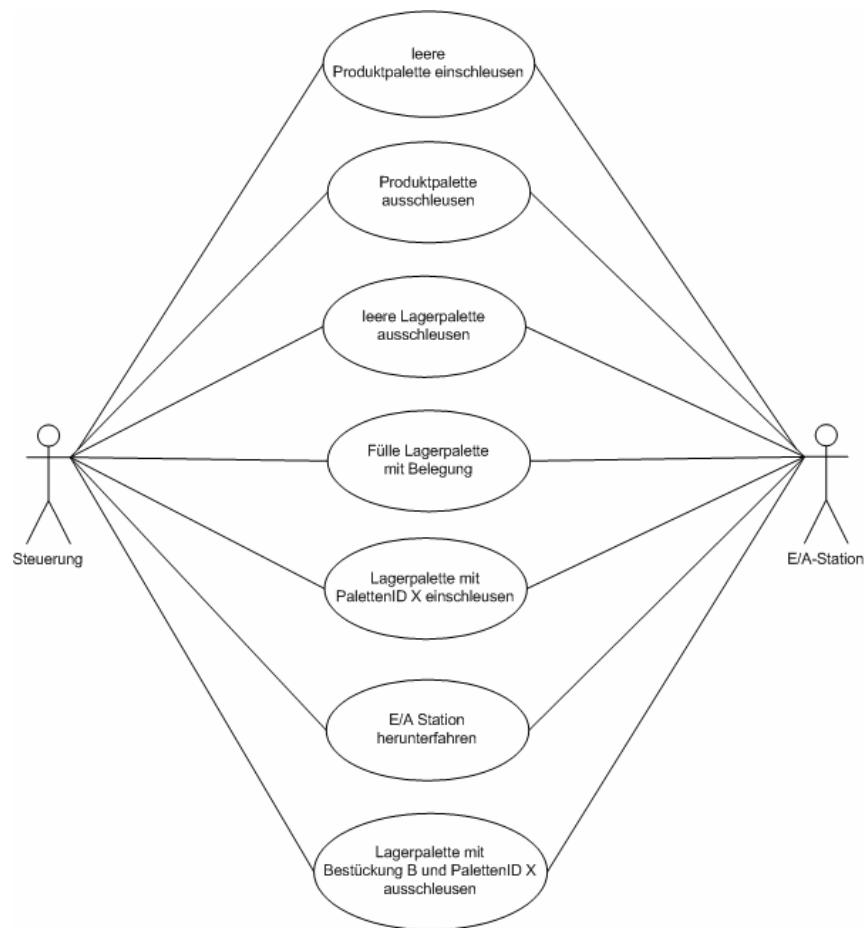


Abbildung 2.61: Steuerung an E/A-Station

Die E/A-StationsSteuerung reagiert auf sieben verschiedene Nachrichten, welche die Steuerung sendet.

Empfängt die E/A-StationsSteuerung den Befehl zum Ausschleusen einer leeren Lagerpalette, wird die Lagerpalette vom Band genommen. Hierzu wird dem Mitarbeiter durch den PC angezeigt, dass er die Palette vom Schlitten auf dem Band nehmen soll. Bestätigt der Mitarbeiter,

2.5 Die Eingabe/Ausgabe-Station (E/A-System)

dass er diese Aktion ausgeführt hat, wird eine Antwort an Steuerung zurückgeschickt.

Beim Empfang einer Anforderung zum Ausschleusen einer Produktpalette wird dem Mitarbeiter mitgeteilt, dass er eine Produktpalette vom Band nehmen soll. Nach der Bestätigung und dem Ausführen des Auftrages durch den Mitarbeiter bestätigt die E/A-StationsSteuerung der Steuerung die Ausführung der Anforderung.

Bekommt die E/A-StationsSteuerung den Befehl, eine Palette mit einer bestimmten Belegung zu füllen, wird dieser Befehl nach unten an die E/A-Station weitergegeben. Nach dem Befüllen bestätigt die unterste Schicht die Ausführung und liefert die Paletten-ID der bestückten Palette und die tatsächliche Bestückung an die E/A-Stationssteuerung zurück. Die E/A-StationsSteuerung schickt danach sofort eine weitere Nachricht mit den gleichen Parametern (Paletten-ID und Bestückung) an die Steuerung, welche das erfolgreiche Befüllen bestätigt.

Um die Anforderung zu erfüllen, eine bestimmte Lagerpalette einzuschleusen, wird dem Mitarbeiter durch den PC mitgeteilt, dass die Lagerpalette mit Paletten-ID xy benötigt wird. Nach dem Aufsetzen der Palette auf den Schlitten und der Bestätigung durch den Mitarbeiter wird ein Acknowledge an Steuerung geschickt (zusammen mit der Paletten-ID und der Belegung der Palette als Parameter).

Wenn eine leere Produktpalette eingeschleust werden soll, wird der Mitarbeiter darüber informiert, dass eine leere Produktpalette auf dem Schlitten vor der E/A-Station benötigt wird. Bestätigt der Mitarbeiter die Ausführung der Anforderung, dann teilt die E/A-StationsSteuerung der Steuerung mit, dass nun eine leere Produktpalette auf dem Schlitten liegt.

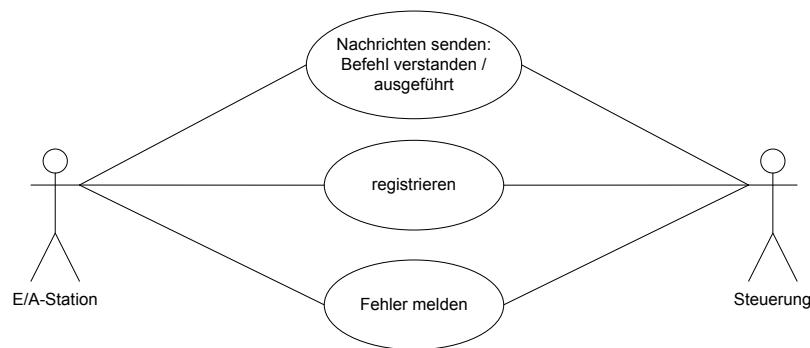
Wenn die Steuerung mitteilt, dass das System heruntergefahren werden soll, wird dem Arbeiter mitgeteilt, dass das System abgeschaltet wird. Anschließend beendet sich die E/A-StationsSteuerung.

Wenn der Befehl "NICHT leere Lagerpalette ausschleusen" von der Steuerung an die E/A-StationsSteuerung übergeben wird, wird diese vom Band genommen. Zudem wird in der Bestandsdatei der E/A-StationsSteuerung die derzeitige Befüllung und die PalettenID dieser Lagerpalette gespeichert, um in der Folge darauf zugreifen zu können und diese zu einem späteren Zeitpunkt mit einer bestimmten Befüllung wieder einführen zu können. Dabei gilt es zu beachten, dass diese Lagerpalette während des Aufenthaltes an der E/A-Station nicht in ihrer Bestückung verändert werden darf, ausser, wenn die E/A-StationsSteuerung von der Steuerung einen Befehl erhält, diese Palette mit einer anderen Bestückung zu befüllen. Andererseits kann die befüllte Palette in der Folge mit der gleichen Farbe, welche sich schon auf der Palette

2.5 Die Eingabe/Ausgabe-Station (E/A-System)

befindet, nachbestückt werden.

2.5.2 E/A-StationsSteuerung an Steuerung



Die E/A-StationsSteuerung hat die Pflicht, sich bei der Steuerung zu registrieren, woraufhin diese informiert ist, dass die E/A-StationsSteuerung einsatzbereit ist.

Die E/A-StationsSteuerung schickt eine Nachricht, dass der Befehl verstanden wurde. Hat der Mitarbeiter eine Palette entnommen oder auf den Schlitten gestellt, bestätigt er durch Betätigung eines Schalters. Erhält die Steuerung E/A-Station diese Benachrichtigung, schickt sie eine Bestätigung an Steuerung, dass das Kommando erfolgreich ausgeführt wurde.

Außerdem muss die E/A-StationsSteuerung bestimmte Fehler, die sie nicht selbst beheben kann, nach oben zur Steuerung melden können.

Teil II

Konstruktion der produktiven Systeme

3 Gemeinsame Module

3.1 Einleitung

Beim Betrachten der Konstruktion wird offensichtlich, dass die Netzwerkkommunikation von jedem Subsystem verwendet wird. Deswegen wurde beschlossen, den Code der gestalt zu schreiben, der von allen Teilen der FDZ verwendet werden kann.

Durch diesen Ansatz wird das Lager mit seinem Betriebssystem RTOS zum limitierenden Faktor, was die Randbedingungen für die Programmierung angeht: Java ist für dieses System nicht verfügbar, und auf dem Aufgrund der limitierten Ressourcen wurde auch vorerst auf den Einsatz von C++ verzichtet.

Das Kommunikationsmodul wurde sich abstützend auf die POSIX-API in C entwickelt.

Um diesen Code auch für das Simulationssystem verfügbar zu machen, wurde ein JNI (Java Native Interface) - Wrapper geschrieben, der es ermöglicht, den selben Code auch aus Java heraus anzusprechen.

Des Weiteren wurde nachträglich eine C++ -Kapselung für die C-Implementierung der Kommunikations-Bibliothek geschrieben. Dieses stellt die Netzwerkfunktionalität objektorientiert zur Verfügung.

Dabei wurde die Netzwerkfunktionalität nicht neu implementiert, sondern die cpp-Klasse ruft intern die Funktionen der c-Kommunikations-Bibliothek auf.

3.2 jniBridge

Um nicht-Java-Code in Java ansprechen zu können, muss dieser in eine Dynamic Loadable Library (für Windows) oder in ein Shared Object (für Linux) verpackt werden.

Dieses Package enthält lediglich eine Klasse, die das Laden der dll bzw. des so übernimmt.

Vergleiche `JNIBridge`, Kapitel ??

3.3 fdzMessageHandler

Der MessageHandler wurde eingeführt, um die Schnittstelle aller Funktionen, die auf Netzwerknachrichten reagieren zu vereinheitlichen: Für jede empfangene Nachricht auf die reagiert werden muss, ist es nötig eine `callback`-Funktion zu registrieren.

Dazu wird die Funktion `registerCallback()` bereitgestellt, die einer Nachricht die entsprechende `callback`-Funktion zuordnet. Alle Details siehe Kapitel ??

Wenn eine Nachricht ankommt, muss geprüft werden, ob ein `callback` für die Nachricht hinterlegt ist. Das passiert in `parseMessage()`, siehe Kapitel ??.

3.4 fdzNetwork

Für die Teile der Software, die in Java implementiert sind, wurde die selbe Funktionalität objektorientiert zu Verfügung gestellt. In Java muss kein callback, sondern ein `JNIMessageCallback` hinterlegt werden. Dieser wird dann an `JNIMessageHandler::registerCallback()` übergeben, siehe Kapitel ??.

Wenn im Java eine Nachricht ankommt, muss `JNIMessageHandler::dispatchMessage()` mit der Nachricht ausgeführt werden. Vergleiche Kapitel ??.

3.4 fdzNetwork

Das Modul fdzNetwork wurde eingeführt, um das handling von TCP / IP auf allen Systemen zu vereinheitlichen. Die Dokumentation zur C-Implementierung gibt es hier: Kommunikations-BibliothekFunk, Kapitel ?. Für die Javaseite gibt es auch hier einen Wrapper, der lediglich die Funktionalität 1 : 1 durchschleift. Die Dokumentation dazu liegt hier: Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung, Kapitel ??.

3.5 Kommunikations-Bibliothek (c-Implementierung)

Die Kommunikations-Bibliothek ist eine allgemeine Library für Netzwerkkommunikation mit TCP/IP, basierend auf dem Client-Server-Modell.

Sie wurde aufbauend auf dem POSIX-Standard entwickelt und ist daher portabel.

Die wichtigsten Aufgaben der Kommunikations-Bibliothek sind:

- Eine einheitliche Schnittstelle für alle Subsysteme des Gesamtsystems bereitstellen.
- Die Unterschiede bei der Implementierung der Kommunikation zwischen den einzelnen Subsystemen verstecken.
- Senden und Empfangen von vollständigen Nachrichten mit Hilfe von mehreren Sockets, jedoch weitestgehend unabhängig vom Nachrichtenprotokoll.

Die Kommunikations-Bibliothek soll von allen Subsystemen verwendet werden. Sie stellt alle notwendigen Operationen zur Verfügung, um die Kommunikation zwischen den Subsystemen zu implementieren. Der Vorteil dabei ist, dass nur einmal Code entwickelt und getestet werden muss. Außerdem müssen Änderungen nur an einer Stelle vorgenommen werden und wirken sich dann auf alle verwendenden Subsysteme aus.

3.5.1 Einbindung der Kommunikationsbibliothek

Die Kommunikationsbibliothek steht entweder direkt als Source-Code zur Verfügung (bei Plattformen, die keine Libraries einbinden können) oder als vorkompilierter Objekt-Code.

Die kompilierte Code-Datei trägt den Namen fdzNetwork und eine plattformabhängige Extension. Für Windows-Plattformen wird die Kommunikationsbibliothek als statische Library kompiliert

3.5 Kommunikations-Bibliothek (c-Implementierung)

(Endung .lib). Gleiches gilt für Unix-Plattformen, wobei die kompilierte Datei hier die Endung .a trägt.

Ein anwendendes Programm muss zur passenden Library linken. Zusätzlich muss natürlich der Haupt-Header der Library, fdzNetwork.h (siehe ??), eingebunden werden, welcher die Schnittstelle zur Verfügung stellt.

3.5.2 Funktionalität der Kommunikations-Bibliothek

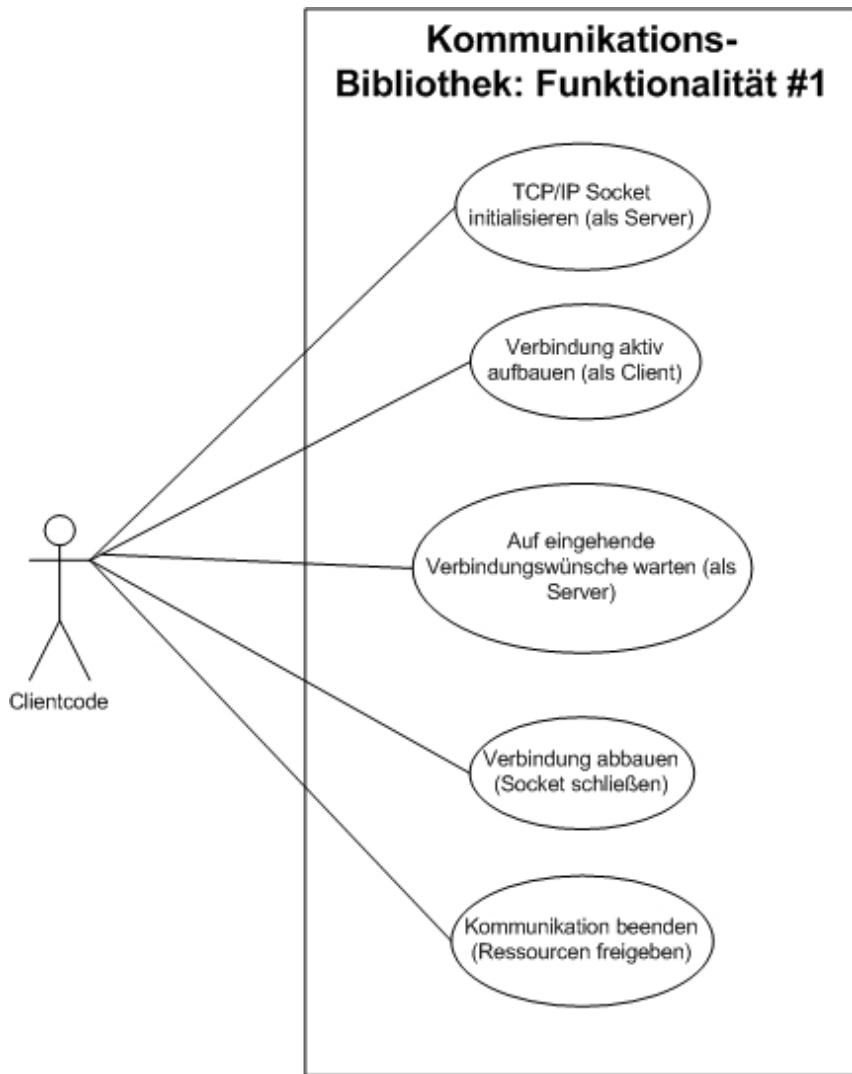


Abbildung 3.1: Funktionalität der Kommunikationsbibliothek

Der Anwender der Kommunikations-Bibliothek hat die Möglichkeit, eine oder mehrere TCP/IP-Verbindungen zu einem anderen System herzustellen bzw. abzubauen (entweder als Client oder Server).

3.5 Kommunikations-Bibliothek (c-Implementierung)

Der Anwender muss am Ende die gesamte Kommunikation beenden, d.h. alle offenen Verbindungen schließen und verwendete Ressourcen freigeben.

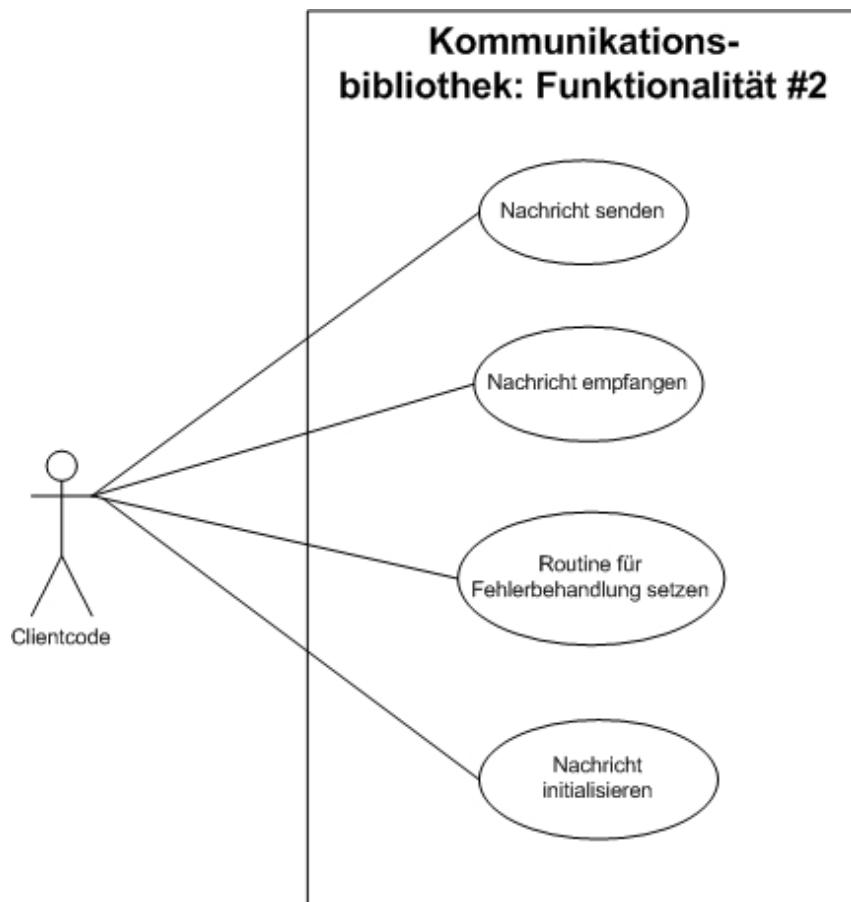


Abbildung 3.2: Funktionalität der Kommunikationsbibliothek

Nach Aufbau der Verbindung ist es möglich, Nachrichten zu senden und zu empfangen.

Nachrichten sollten vor dem Senden mit einem String initialisiert werden.

Darüber hinaus werden Mechanismen zur Fehlerbehandlung bereitgestellt. Der Anwender kann eine eigene Programmroutine festlegen, die im Falle eines Netzwerkfehlers aufgerufen wird.

Auf Funktionalität zur Prüfung der Nachricht auf Korrektheit (entsprechend eines zugrundeliegenden Protokolls) wurde verzichtet, weil die Bibliothek unabhängig vom Protokoll arbeiten soll.

3.5 Kommunikations-Bibliothek (c-Implementierung)

3.5.3 Verwendung der Schnittstelle

3.5.3.1 Clientverbindung

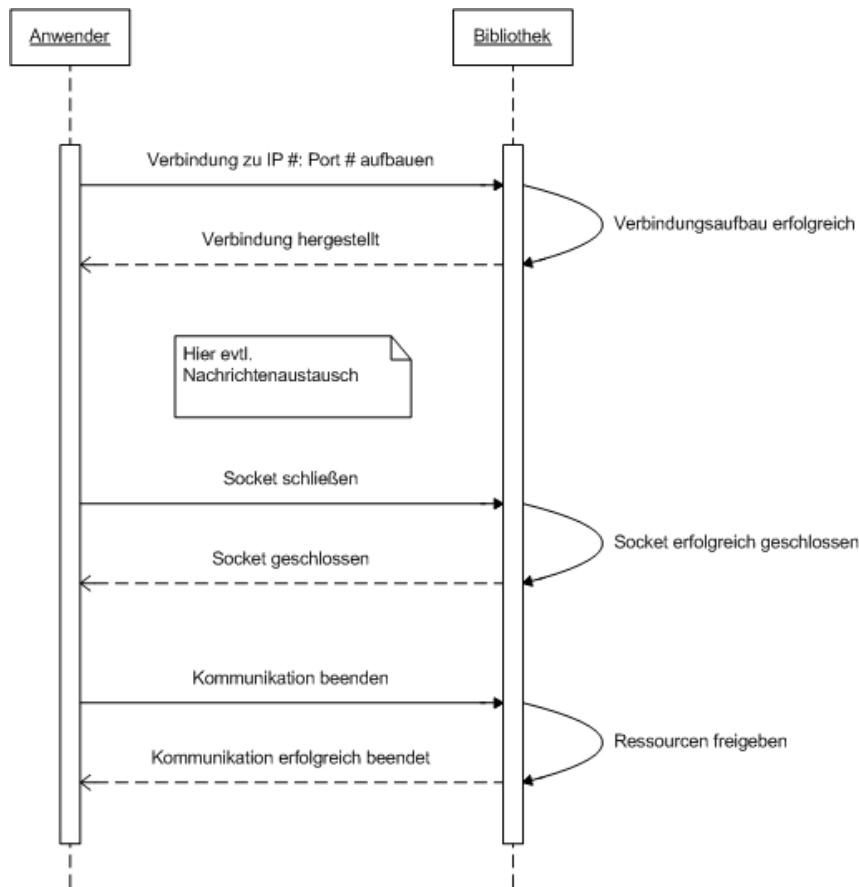


Abbildung 3.3: Anwendung als Client

Wenn man die Kommunikations-Bibliothek als Client verwenden möchte, muss man zunächst eine Verbindung zum Server aufbauen (Übergabe von IP-Adresse und Port-Nummer erforderlich). Man kann dann Nachrichten versenden und empfangen. Sind diese Vorgänge abgeschlossen, muss die Verbindung abgebaut werden, wobei der zu Beginn erzeugte Socket wieder geschlossen wird. Zusätzlich müssen verwendete Ressourcen freigegeben werden.
Das Diagramm zeigt nur die Abläufe für die Kommunikation über eine Verbindung.

3.5 Kommunikations-Bibliothek (c-Implementierung)

3.5.3.2 Serververbindung

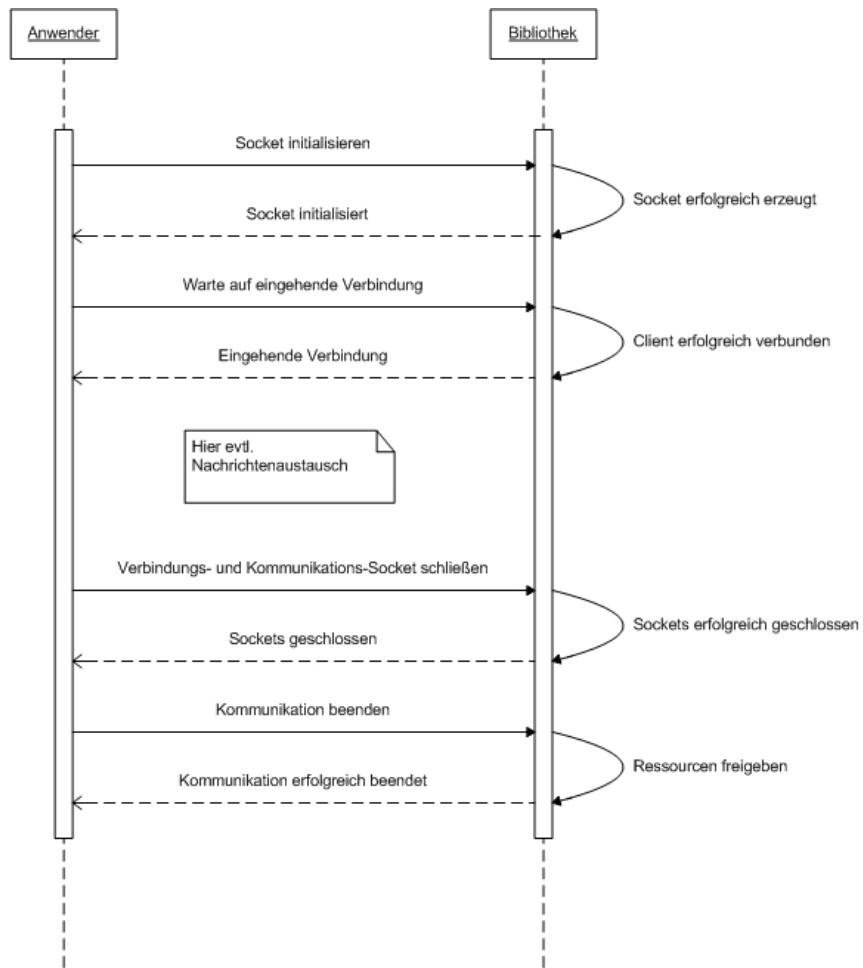


Abbildung 3.4: Anwendung als Server

Bei der Verwendung als Server muss zunächst ein Socket initialisiert werden. Anschließend muss der Anwender die Kommunikationsbibliothek dazu verwenden, auf eingehende Verbindungen zu warten. Verbindet sich ein Client auf den Socket, so ist der Datenaustausch über die Kommunikations-Bibliothek möglich, allerdings auf einem anderen Socket. Der Anwender muss nach Beendigung der Kommunikations-Aktivitäten beide Sockets (Verbindungs- und Kommunikationssocket) sowie verwendete Ressourcen freigeben.

3.5 Kommunikations-Bibliothek (c-Implementierung)

3.5.3.3 Nachrichten senden

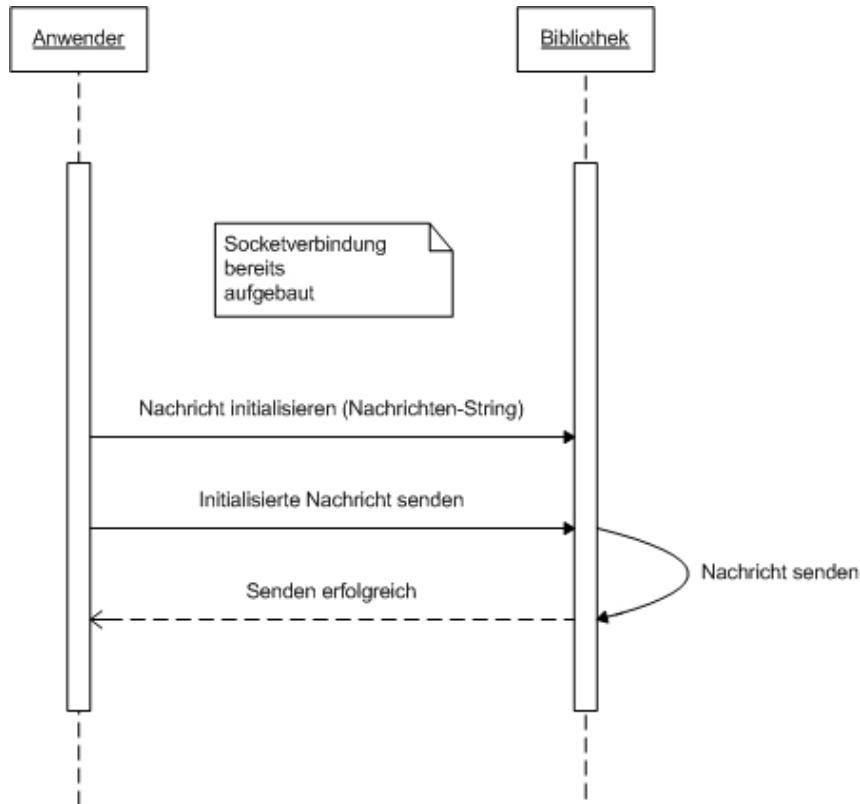


Abbildung 3.5: Senden einer Nachricht

Ist bereits eine Socketverbindung zwischen zwei Kommunikationsteilnehmern aufgebaut, kann ein Teilnehmer Nachrichten in Form eines Byte-Stroms senden. Nach dem Senden einer vollständigen Nachricht erhält der Sender die Bestätigung der erfolgreichen Durchführung. Im Fehlerfall erhält er eine Fehlerbenachrichtigung.

3.5 Kommunikations-Bibliothek (c-Implementierung)

3.5.3.4 Nachrichten empfangen

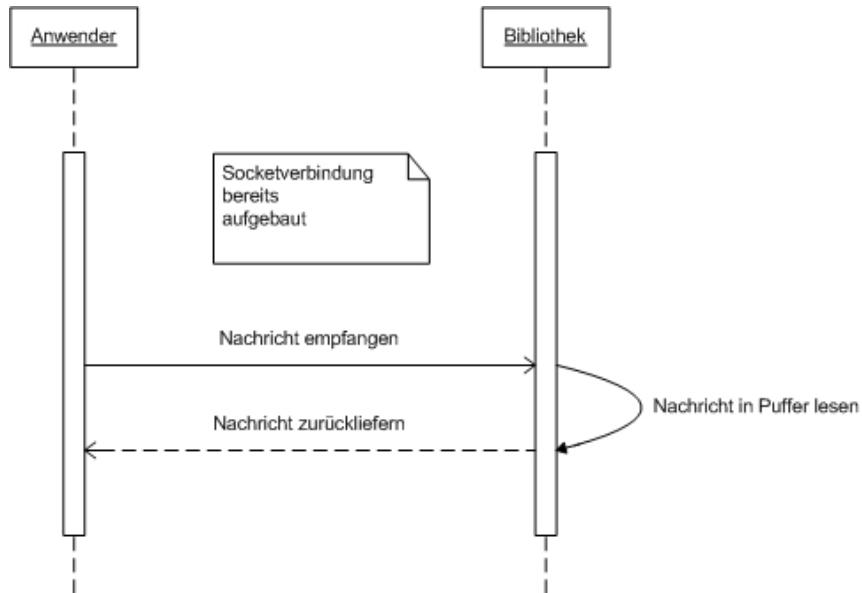


Abbildung 3.6: Empfangen einer Nachricht

Ist bereits eine Socketverbindung zwischen zwei Kommunikationsteilnehmern aufgebaut, verwendet der Anwender eine blockierende Funktion, um eine Nachricht zu empfangen. Die Funktion liefert die empfangenen Bytes einer vollständigen Nachricht und wird erst dann beendet.

3.5.4 Fehlerbehandlung

Der Anwender der Kommunikations-Bibliothek wird durch Rückgabe-Nachrichten der verwendeten Funktionen auf Fehler aufmerksam gemacht. Es gibt einen einzigen reservierten Wert, der einen Fehler einer Routine der Kommunikations-Bibliothek angezeigt.

Zusätzlich hat der Anwender die Möglichkeit, eine Funktion anzugeben, die im Falle eines Fehlers immer vor dem Ende der entsprechend aufgerufenen Funktion aufgerufen wird. Der Anwender bekommt hierbei detaillierte Informationen über die Art des aufgetretenen Fehlers mitgeliefert und kann entsprechende Maßnahmen ergreifen.

3.5.5 Funktions-Prototypen

Listing 3.1: Header der Kommunikations-Bibliothek

```
/*
 ****
 */
/*===== SMessage-Struktur
```

3.5 Kommunikations-Bibliothek (c-Implementierung)

```
*****  
*/  
typedef struct _SMessage  
{  
    char message [MAX_MESSAGELENGTH];  
    unsigned short int len;  
    char* payload;  
} SMessage;  
  
#ifdef WIN32  
    typedef int socklen_t;  
#endif /*win32*/  
  
/*  
*****  
*/  
/*===== Error-Handler-Prototyp  
*****  
*/  
typedef int (*HandlerType) (int connID, int err_no, const char* msg,  
    const char* err_no_msg);  
  
/*  
*****  
*/  
/* ====== Funktionsprototypen */  
/*  
*****  
*/  
  
#ifndef RTOS    /*Schliesst Serverfunktionen unter RTOS aus*/  
/******initServerSocket*****/  
int initServerSocket(unsigned short port);  
/*******/  
  
/******awaitConnection*****/  
int awaitConnection(int sockfd, int maxConnections);  
/*******/  
#endif /*rtos*/  
  
/******openConnection*****/  
int openConnection(const char* ip, unsigned short port);  
/*******/  
  
/******sendMessage*****/  
int sendMessage(int fd, SMessage* message);  
/*******/  
  
/******receiveMessage*****/  
int receiveMessage(int fd, SMessage* message);  
/*******/
```

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

```
*****initMessage*****
void initMessage(SMessage* message, const char* msg_string);
*****
```



```
*****initMessagePayloadLength*****
void initMessagePayloadLength(SMessage* m);
*****
```



```
*****closeSocket*****
int closeSocket(int fd);
*****
```



```
*****cleanup*****
int cleanup(void);
*****
```



```
*****setErrorHandler*****
void setErrorHandler(HandlerType handler);
*****
```

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

Die cpp-Kommunikations-Bibliothek stellt zwar im Grunde nur eine objektorientierte Kapselung der bisherigen Kommunikations-Bibliothek dar, allerdings mit folgenden Besonderheiten:

- der ErrorHandler kann wie zuvor benutzt werden, jedoch ist dieser nur in den c-Funktionen der Kommunikations-Bibliothek aktiv. Fehler im cpp-Teil werden nicht gemeldet.
- die cpp-Kommunikationsbibliothek ersetzt die c-Implementierung nicht, sondern baut auf dieser auf
- die Unabhängigkeit vom Nachrichtenprotokoll ist nicht mehr, bzw. nur noch sehr begrenzt gegeben
- da Teile der Recovery-Funktionalität mit implementiert sind, ist die cpp-Kommunikations-Bibliothek nur begrenzt für andere Subsysteme einsetzbar

3.6.1 Einbindung der Kommunikationsbibliothek

Die cpp-Kommunikations-Bibliothek steht entweder als Source-Code zur Verfügung (bei Plattformen, die keine Libraries einbinden können) oder als vorkompilierter Objekt-Code.

Die Kompilierte Code-Datei trägt den Namen fdzNetworkcpp und eine plattformabhängige Extension.

Zusätzlich zur Library muss natürlich noch der Haupt-Header der Library, fdzNetwork.hpp, eingebunden werden, welcher die Schnittstellen zu Verfügung stellt.

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

3.6.2 Funktionalität der Kommunikations-Bibliothek

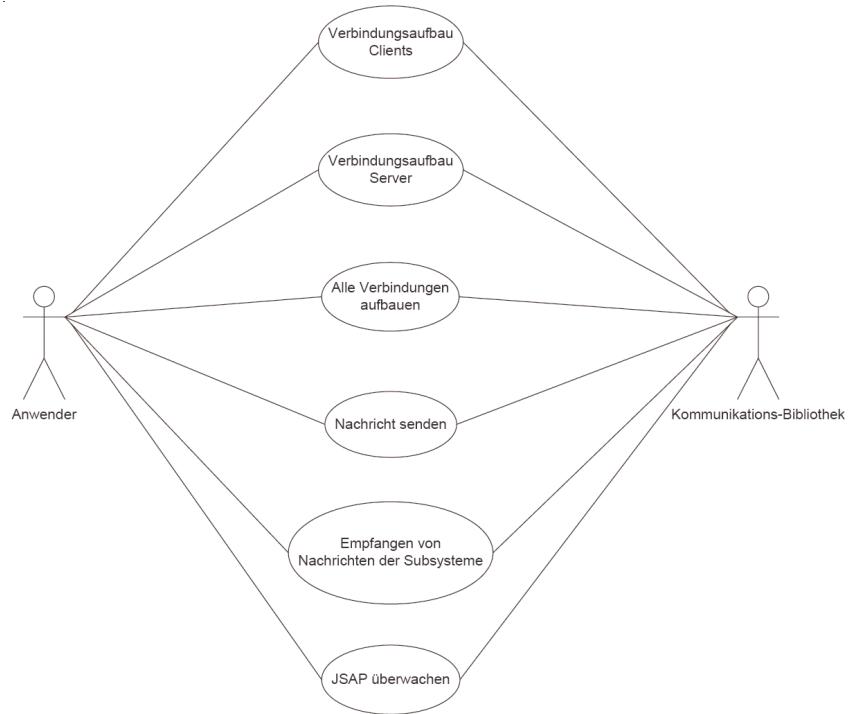


Abbildung 3.7: Funktionalität der Kommunikationsbibliothek

Der Anwender hat die Möglichkeit beliebig viele Verbindung als Server und / oder Client aufzubauen. Auf aktiven Verbindungen können Nachrichten verschickt oder empfangen werden. Außerdem können die Verbindungen blockierend oder nicht-blockierend überwacht werden ob Nachrichten angekommen sind.

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

3.6.3 Verwendung der Schnittstelle

3.6.3.1 Clientverbindung

sd Verbindungsauflauf Client:

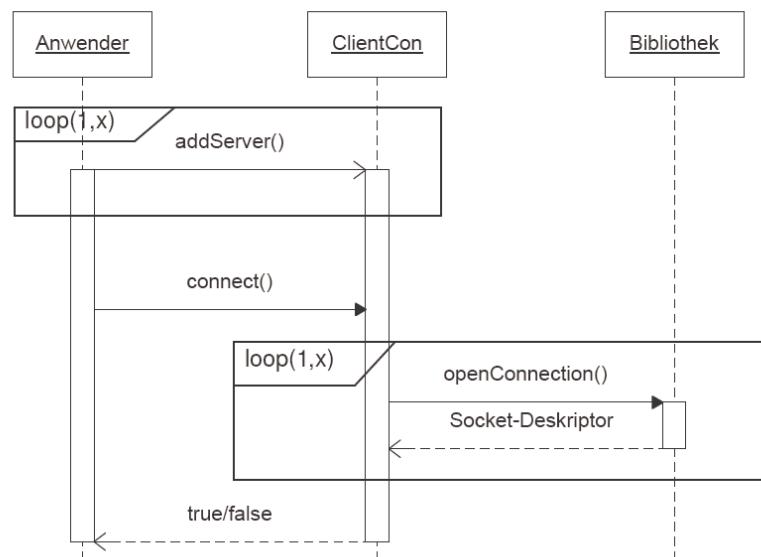


Abbildung 3.8: Anwendung als Client

Es können beliebig viele Server mit der Funktion `addServer()` registriert werden. Mit `connect()` werden alle Serververbindungen aufgebaut in der Reihenfolge wie sie registriert wurden.

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

3.6.3.2 Serververbindung

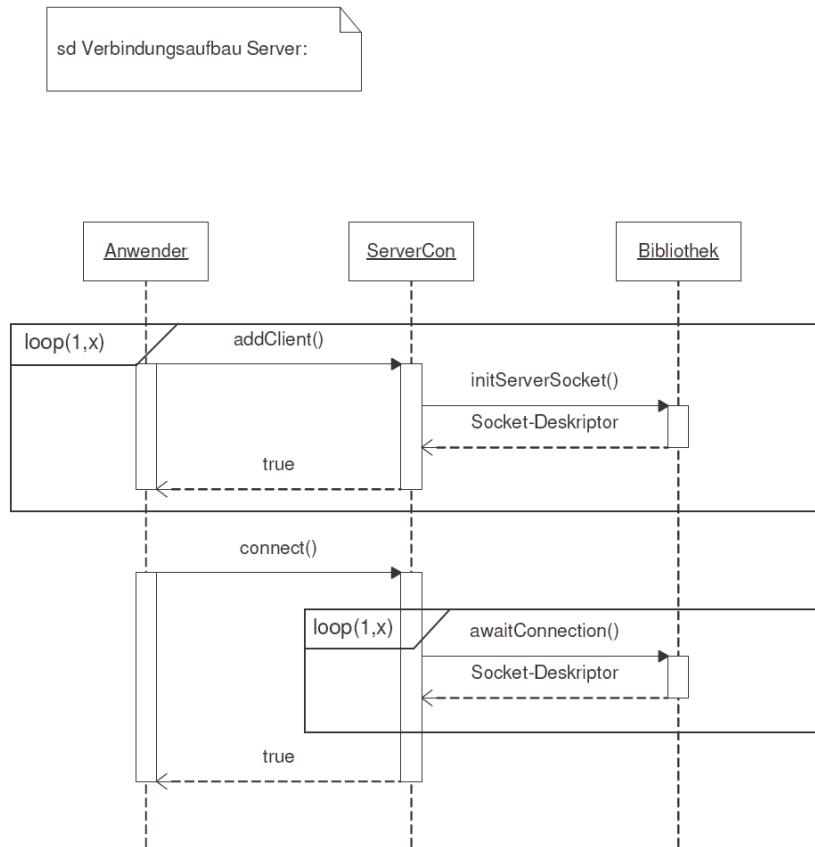


Abbildung 3.9: Anwendung als Server

Es können beliebig viele Clients mit der Funktion `addClient()` registriert werden. Mit `connect()` werden alle Clientverbindungen aufgebaut in der Reihenfolge wie sie registriert wurden.

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

3.6.3.3 Nachrichten senden

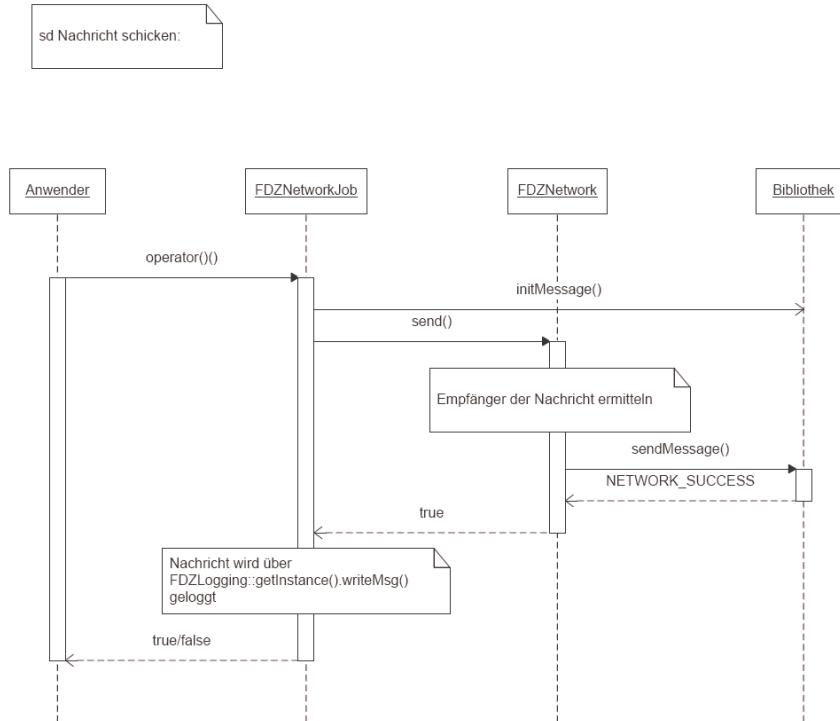


Abbildung 3.10: Senden einer Nachricht

Um Nachrichten zu schicken wird der `()`-Operator der Klasse `FDZNetworkJob` aufgerufen. Aus der Nachricht wird der Empfänger ermittelt und diese über die entsprechende Verbindung gesendet.

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

3.6.3.4 Nachrichten empfangen

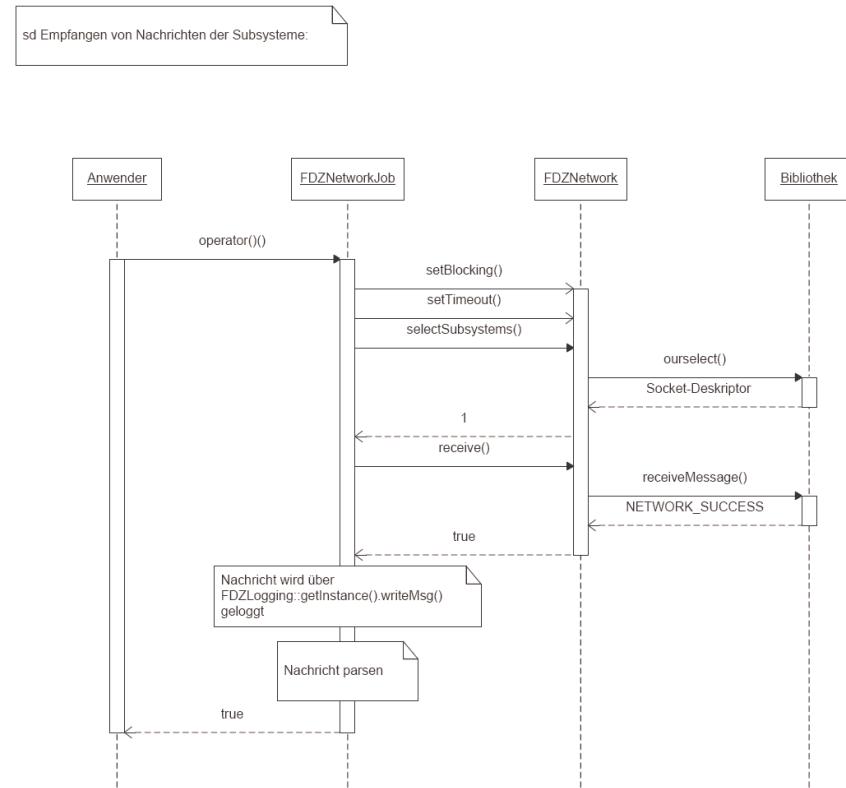


Abbildung 3.11: Empfangen einer Nachricht

Um Nachrichten zu empfangen wird der ()-Operator der Klasse FDZNetworkJob aufgerufen. Intern wird der Socket ermittelt auf dem empfangen werden soll und anschliessend die Nachricht abgeholt.

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

3.6.3.5 Server überwachen

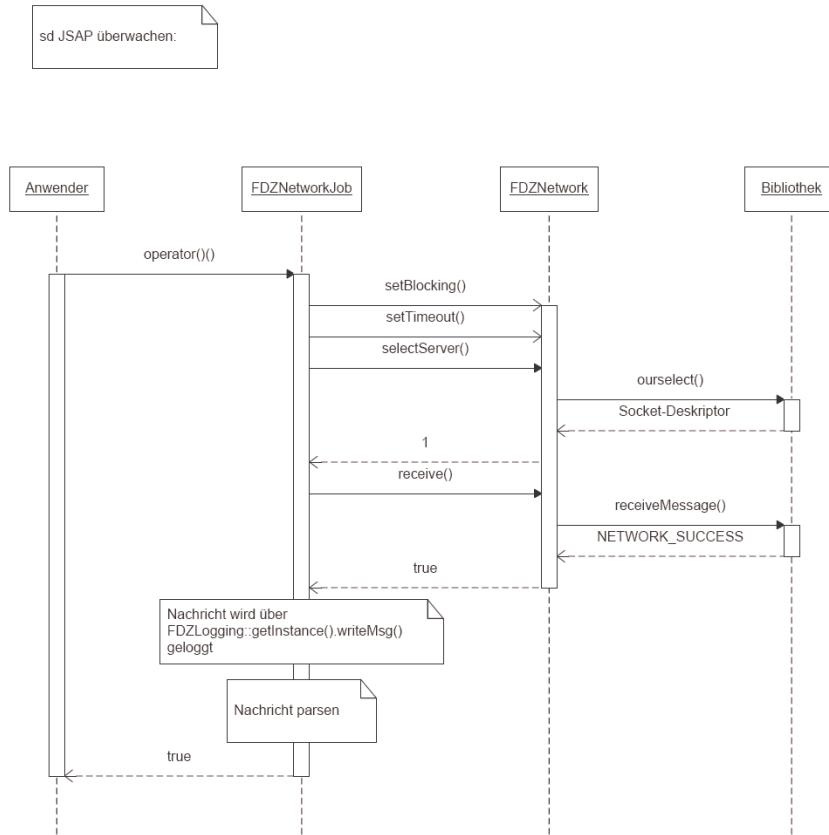


Abbildung 3.12: Verbindungen zu Server überwachen

Um die Serververbindungen zu überwachen, wird der ()-Operator der Klasse FDZNetworkJob.

3.6 Kommunikations-Bibliothek (cpp-Kapselung)

3.6.3.6 Alle Verbindungen aufbauen

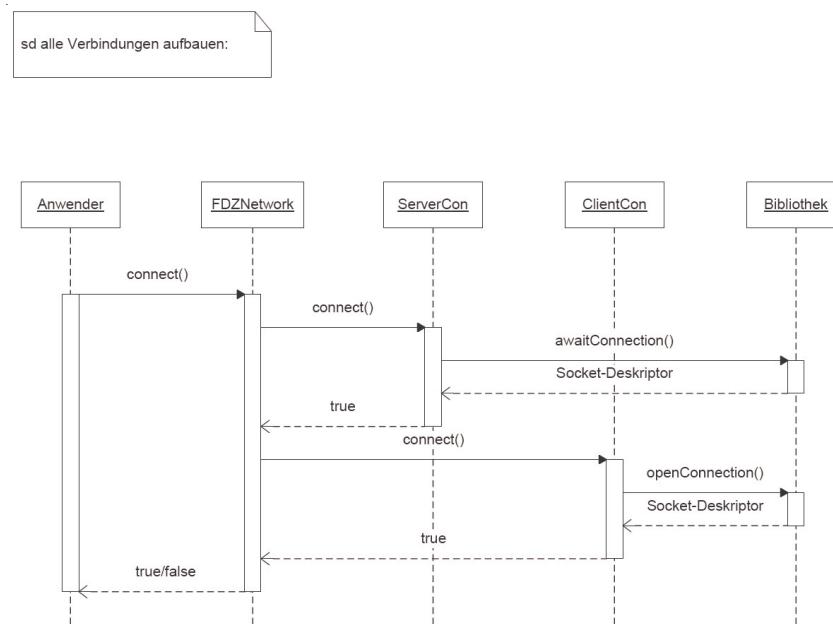


Abbildung 3.13: Alle Verbindungen aufbauen

Um alle Verbindungen für die registrierten Clients und Server aufzubauen wird die Funktion `connect()` der Klasse FDZNetwork aufgerufen. Dabei ist zu beachten, dass die Verbindungsreihenfolge der Registrierreihenfolge entspricht, wobei zuerst alle Clientverbindungen und danach alle Serververbindungen aufgebaut werden.

3.6.4 Fehlerbehandlung

Die meisten Methoden der FDZNetwork-Klasse liefern boolesche Werte als Ergebnis, anhand derer Fehler erkannt und anschließend behandelt werden können.

Der ErrorHandler kann ebenfalls gesetzt und verwendet werden, allerdings wird dieser nur bei den Funktionen der c-Kommunikation-Bibliothek aktiv. Fehler die im cpp-Teil auftreten lösen nicht den ErrorHandler aus.

4 Ablauf der Teilsysteme

4.1 Steuerung

4.1.1 Abläufe innerhalb der Steuerung

4.1.1.1 Beschreibung

Die Steuerung ist verantwortlich für die planmäßige Abarbeitung der vom JSAP erhaltenen Aufträge. Zu Beginn eines Auftrags, muss im Lager angefragt werden, ob ein ausreichender Bestand an Smarties jeder Farbe zur Verfügung steht. Nachdem das Lager den Bestand für jede Farbe bestätigt hat, wird eine Meldung (Auftrag produzierbar) an das JSAP gesendet. Ist der Bestand mindestens einer Farbe nicht ausreichend, wird JSAP mitgeteilt, dass der Auftrag nicht produzierbar ist.

Vor dem Senden eines jedem Kommandos an ein Subsystem muss überprüft werden, ob das JSAP in der Zwischenzeit das Kommando zum Abbruch eines Auftrages an die Steuerung gesendet hat. Ist dies der Fall müssen alle Komponenten in die Ausgangsposition zurückgebracht werden. Das heißt, Lagerpaletten werden zurück ins Lager, Produktpaletten zur E/A Station gefahren und alle Schlitten freigegeben.

Um das Kommando zum Abbruch eines Auftrages empfangen zu können, ist es nötig vor der Abarbeitung der Unteraufträge Nachzufragen, ob JSAP den Befehl zum Abbruch gesendet hat.

Nach dem Abarbeiten aller für einen Auftrag benötigten Teilkommmandos in den Subsystemen, bzw. wenn im Falle eines Abbruchkommandos alle Komponenten in ihre Ausgangsposition zurückgebracht wurden, ist ein Auftrag erfolgreich ausgeführt worden. Dies wird dem JSAP mit der Nachricht Befehl ausgeführt mitgeteilt.

Die nachfolgenden Tabellen und Diagramme beschreiben den Ablauf eines Bestückungsauftrags der Fabrik der Zukunft. Der Ablauf kann für jeden beliebigen Auftrag jeglicher Matrix- und Farbkombinationen angewandt werden.

Die im Ablaufplan dargestellte Schleife läuft über alle Farben, die im Auftrag enthalten sind. Weiterhin kann es dazu kommen, dass die Schleife für eine Farbe mehrfach ausgeführt werden muss, wenn auf einer Lagerpalette der jeweiligen Farbe nicht genügend Smarties für den Auftrag vorhanden sind.

Die Steuerung gleicht beim Hochfahren des Systems die lokale Bestandsdatei mit dem aktuellen Lagerbestand ab. Nun können folgende 4 Fälle auftreten:

Keine Bestandsdatei/kein Recovery Diese Kombination tritt beim **ersten** Hochfahren des Systems auf. Nun muss die Bestandsdatei erstellt werden. Dazu wird wie in Bestandsdatei dargestellt der Lagerbestand abgefragt und in der Bestandsdatei festgehalten.

4.1 Steuerung

Bestandsdatei/kein Recovery Tritt im Normalfall beim Hochfahren des Systems auf. Nun wird die vorhandene Bestandsdatei mit den tatsächlichen Werten des Lagers verglichen. Unterscheiden sich die beiden Werten voneinander, wird der Operator benachrichtigt.

Bestandsdatei/Recovery Dieser Fall tritt ein, wenn im Vorfeld beim Bearbeiten eines Auftrages ein Fehler eingetreten ist. Nun muss der Operator entscheiden, ob der vorher abgebrochene Auftrag fortgeführt oder beendet werden soll.

Keine Bestandsdatei/Recovery Fataler Fehler! Wie im obigen Fall kam es auch hier im Vorfeld zu einem Fehler. Da jedoch die Bestandsdatei fehlt, muss der Auftrag abgebrochen werden und das System neu initialisiert werden.

4.1.1.2 Ablauf für Bestückungsauftrag

Die nachfolgenden Diagramme beschreiben den Ablauf eines Bestückungsauftrags der Fabrik der Zukunft. Der Ablauf kann je nach Befüllungsstand des Lagers unterschiedlich ablaufen. Deshalb ist der Ablauf in 7 statische Teilabläufe zerteilt. Alle Diagramme über die folgenden 7 Teilabläufe verzichten zur vereinfachten Darstellung auf die Antworten der Subsysteme (ACK001 und ACK002).

4.1.1.2.1 Produktpalette holen (QUEUE_1)

Der erste Teilablauf beschreibt den Vorgang der Überprüfung des benötigten Bestandes und des Bereitstellens einer Produktpalette beim Roboter.

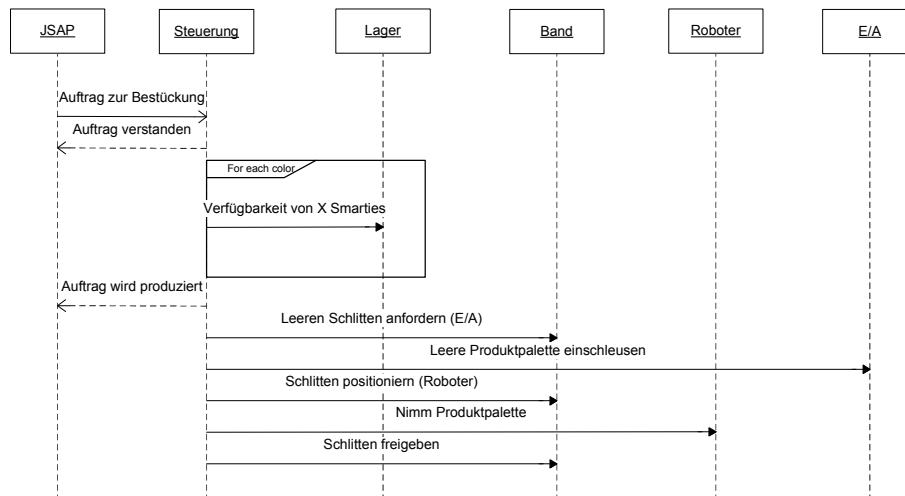


Abbildung 4.1: Queue Produktpalette holen

Zu Beginn eines jeden Auftrages muss im Lager für jede Farbe angefragt werden, ob der Bestand für den Auftrag ausreichend ist. Im Gutfall wird dem JSAP mitgeteilt, dass der Auftrag produzierbar ist, sonst wird Auftrag nicht produzierbar an JSAP gesendet. Anschließend muss zuerst eine leere Produktpalette von der E/A - Station bereitgestellt werden. Diese Palette wird

4.1 Steuerung

dann vom Band an den Roboter gefahren. Dann wird die Produktpalette vom Roboter auf den Bestückungsplatz gestellt.

Befehl	Subsystem	Parameter	Rückgabe
-Schleife über alle Farben-			
Verfügbarkeit von X Smarties der Farbe Y	LAGER	Farbe/Anzahl	
-Schleifenende-			
Leeren Schlitten anfordern(E/A)	BAND		SchlittenID
Leere Produktpalette einschleusen	E/A		
Schlitten positionieren(ROBOTER)	BAND	SchlittenID	
Nimm Produktpalette	ROBOTER		
Schlitten freigeben	BAND	SchlittenID	

Tabelle 4.1: Queue Produktpalette holen

4.1.1.2.2 Lagerpalette ausschleusen (QUEUE_2)

Dieser Teilablauf beschreibt das Ausschleusen einer Lagerpalette aus dem Lager.

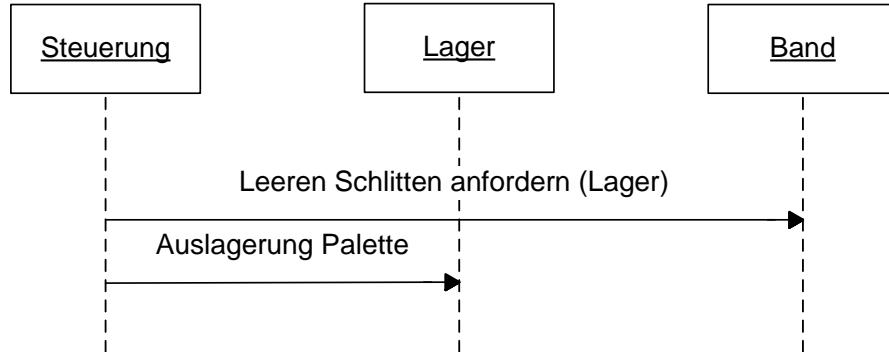


Abbildung 4.2: Queue Lagerpalette ausschleusen

Diese beiden atomaren Teilschritte sind in einem separaten Teilablauf zusammengefasst, da erst nach dem Auslagern der Palette die Anzahl der tatsächlich auf der Lagerpalette befindlichen Smarties bekannt ist. Dadurch ist der nächste Schritt des Bestückungsvorgangs definiert. Wird im Folgenden die Lagerpalette am Roboter nicht leer, so beginnt Teilablauf 3a. Wird die Lagerpalette durch den Bestückungsvorgang am Roboter komplett leer, wird der alternative Ablauf 3b vorgesehen.

4.1.1.2.3 Bestückung von dieser Lagerpalette beendet - Palette nicht leer (QUEUE_3a)

Dieser weitere Teilablauf beschreibt die Bestückung der Produktpalette, wenn während des Bestückungsvorgangs die aktuelle Lagerpalette **mit Restbestand** zurück in das Lager zurück

4.1 Steuerung

Befehl	Subsystem	Parameter	Rückgabe
Leeren Schlitten anfordern(LAGER)	BAND		SchlittenID
Auslagerung Palette	LAGER	Farbe/Anz	Matrix,Anz,PallID

Tabelle 4.2: Queue Lagerpalette ausschleusen

gefahren werden soll.

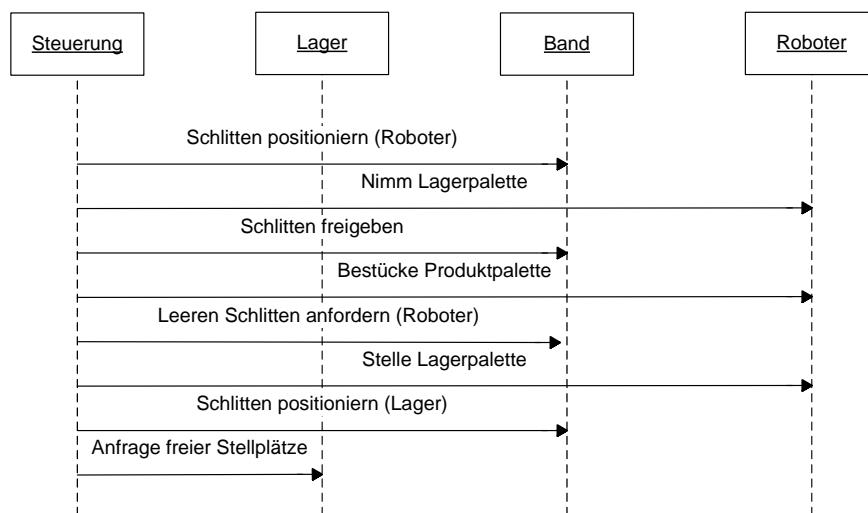


Abbildung 4.3: Queue Bestückung von dieser Lagerpalette beendet - Palette nicht leer

Die Lagerpalette wird vom Lager aus zum Roboter gefahren. Der Roboter stellt sie auf einen der beiden Arbeitsplätze. Im Weiteren füllt der Roboter die Produktpalette mit Smarties der auf der Lagerpalette befindlichen Farbe. Der Roboter beendet die Befüllung der Produktpalette der aktuellen Farbe, wenn alle für den Auftrag notwendigen Smarties von der Lagerpalette genommen wurden. Auf der Lagerpalette befindet sich wie in Kapitel Palette ausschleusen beschrieben, mindestens ein Smartie. Dieser Teilschritt endet mit dem Transport der Lagerpalette zum Lager und der Anfrage auf einen freien Stellplatz. Ist noch ein Stellplatz im Lager verfügbar, wird mit Teilablauf Lagerpalette einlagern fortgefahrt. Sonst folgt Teilablauf Lagerpalette ausschleusen.

4.1.1.2.4 Bestückung von dieser Lagerpalette beendet - Palette leer(QUEUE_3b)

Dieser weitere Teilablauf beschreibt die Bestückung der Produktpalette, wenn während des Bestückungsvorgangs die aktuelle Lagerpalette **ohne Restbestand** zur E/A - Station zur Wiederbefüllung gefahren werden soll.

Die Lagerpalette wird vom Lager aus zum Roboter gefahren. Der Roboter stellt sie auf einen der beiden Arbeitsplätze. Im Weiteren füllt der Roboter die Produktpalette mit Smarties der auf der Lagerpalette befindlichen Farbe. Der Roboter beendet die Befüllung der Produktpalette der aktuellen Farbe, wenn die Lagerpalette vollständig entleert wurde. Nun wird die leere Lagerpalette zur E/A - Station transportiert, um dort wiederbefüllt zu werden. Dieser Teilschritt endet mit dem

4.1 Steuerung

Befehl	Subsystem	Parameter	Rückgabe
Schlitten positionieren(ROBOTER)	BAND	SchlittenID	
Nimm Lagerpalette	ROBOTER	Matrix	
Schlitten freigeben	BAND	SchlittenID	
Bestücke Produktpalette	ROBOTER	Matrix	
Leeren Schlitten anfordern(Roboter)	BAND		SchlittenID
Gib Lagerpalette	ROBOTER		
Schlitten positionieren(Lager)	BAND	SchlittenID	
Anfrage freier Stellplätze	LAGER		

Tabelle 4.3: Queue Bestückung von dieser Lagerpalette beendet - Palette nicht leer

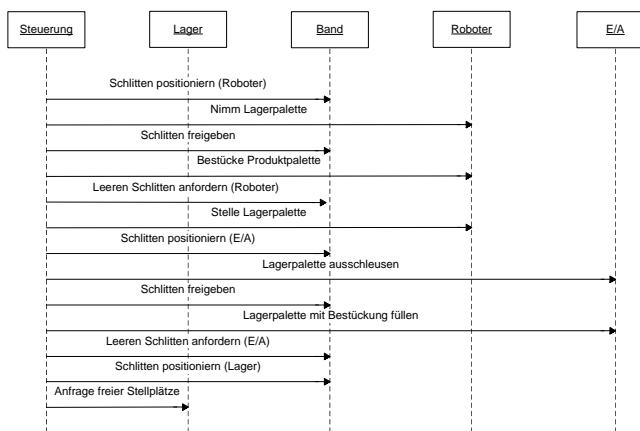


Abbildung 4.4: Queue Bestückung von dieser Lagerpalette beendet - Palette leer

Transport der Lagerpalette zum Lager und der Anfrage auf einen freien Stellplatz. Ist noch ein Stellplatz im Lager verfügbar, wird mit Teilablauf Lagerpalette einlagern fortgefahrene. Sonst folgt Teilablauf Lagerpalette ausschleusen.

4.1.1.2.5 Lagerpalette einlagern (QUEUE_4a)

Folgender Ablauf findet statt wenn mindestens ein freier Stellplatz im Lager zur Verfügung steht.

Steht mindestens ein freier Stellplatz im Lager zur Verfügung, wird die Lagerpalette im Lager eingelagert.

4.1.1.2.6 Lagerpalette ausschleusen (QUEUE_4b)

Steht kein Lagerplatz für die aktuelle Palette zu Verfügung, muss die Lagerpalette zur E/A - Station transportiert, um dort ausgeschleust zu werden.

4.1 Steuerung

Befehl	Subsystem	Parameter	Rückgabe
Schlitten positionieren(ROBOTER)	BAND	SchlittenID	
Nimm Lagerpalette	ROBOTER	Matrix	
Schlitten freigeben	BAND	SchlittenID	
Bestücke Produktpalette	ROBOTER	Anzahl,Farbe	
Leeren Schlitten anfordern(Roboter)	BAND		SchlittenID
Gib Lagerpalette	ROBOTER		
Schlitten positionieren(E/A)	BAND	SchlittenID	
Leere Lagerpalette ausschleusen	E/A		
Schlitten freigeben	BAND	SchlittenID	
Lagerpalette mit Bestückung befüllen	E/A	Matrix	PalettenID
Leeren Schlitten anfordern(E/A)	BAND		SchlittenID
Lagerpalette mit PalettenID einschleusen	E/A	PalettenID	
Schlitten positionieren(Lager)	BAND	SchlittenID	
Anfrage freier Stellplätze	LAGER		

Tabelle 4.4: Queue Bestückung von dieser Lagerpalette beendet - Palette leer

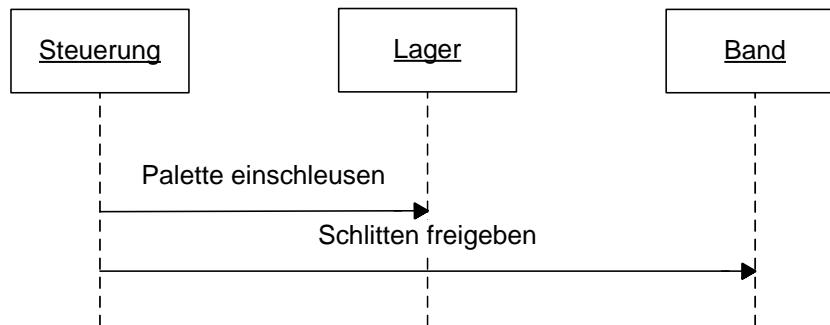


Abbildung 4.5: Queue Lagerpalette einlagern

Wenn kein Lagerplatz für die aktuelle Lagerpalette zu Verfügung steht, muss diese zur E/A - Station transportiert, um dort ausgeschleust zu werden.

4.1.1.2.7 Produktpalette ausschleusen (QUEUE_5)

Am Ende eines jeden Auftrages muss auch die Produktpalette an der E/A - Station ausgeschleust werden.

Die Produktpalette wird vom Roboter aus zur E/A - Station transportiert, um dort ausgeschleust zu werden. Der gesamte Ablauf ist jetzt beendet.

4.1 Steuerung

Befehl	Subsystem	Parameter	Rückgabe
Palette einlagern	LAGER	PalettenID	
Schlitten freigeben	BAND	SchlittenID	

Tabelle 4.5: Queue Lagerpalette einlagern

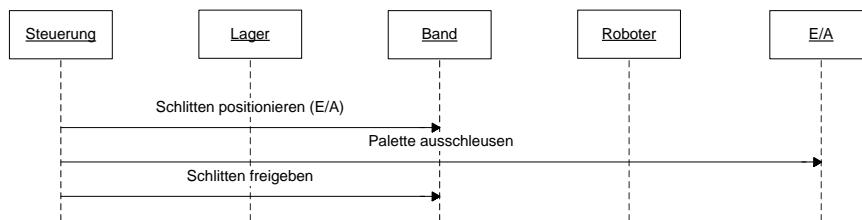


Abbildung 4.6: Queue Lagerpalette ausschleusen

4.1.1.2.8 Ablaufplan für einen Bestückungsauftrag

4.1.1.3 Ablauf für Bestandsabfrage

Dieses Diagramm beschreibt den Ablauf für eine Bestandsabfrage.

Kommt von JSAP das Kommando für eine Bestandsabfrage, so muss für jede einzelne Farbe im Lager der Bestand an Smarties abgefragt werden. Anschliessend werden die erhaltenen Werte mit der Bestandsdatei abgeglichen. JSAP erhält dann den Gesamtbestand an Smarties aller Farben.

4.1.1.4 Ablauf für Auftragsabbruch

JSAP kann zu jeder Zeit das Kommando zum Abbruch des Auftrages senden. Wie in Kapitel ?? beschrieben, muss vor dem Senden eines jedem Kommandos an ein Subsystem überprüft werden, ob das JSAP in der Zwischenzeit das Kommando zum Abbruch eines Auftrages an die Steuerung gesendet hat. Ist dies der Fall müssen alle Komponenten in die Ausgangsposition zurückgebracht werden. Das heißt, Lagerpaletten werden zurück ins Lager, Produktpaletten zur E/A Station gefahren und alle Schlitten freigegeben.

4.1.1.5 Bestandsdatei

4.1.1.5.1 Beschreibung

Die Bestandsdatei, `c_lagerbestand.txt`, stellt die Konsistenz zwischen Steuerung und Lagersystem sicher. Die Datei wird wie in Kapitel ?? beschrieben beim Hochfahren des Systems erstellt bzw. abgeglichen. Weiterhin wird nach jedem vollendeten Auftrag ein Abgleich mit dem Lager durchgeführt. Sollte es zu während des Ablaufs zu Konflikten kommen, so wird die Bestandsdatei der Steuerung überschrieben.

4.1 Steuerung

Befehl	Subsystem	Parameter	Rückgabe
Schlitten positionieren(E/A)	BAND	SchlittenID	
Lagerpalette ausschleusen	E/A		
Schlitten freigeben	BAND	SchlittenID	

Tabelle 4.6: Queue Lagerpalette ausschleusen

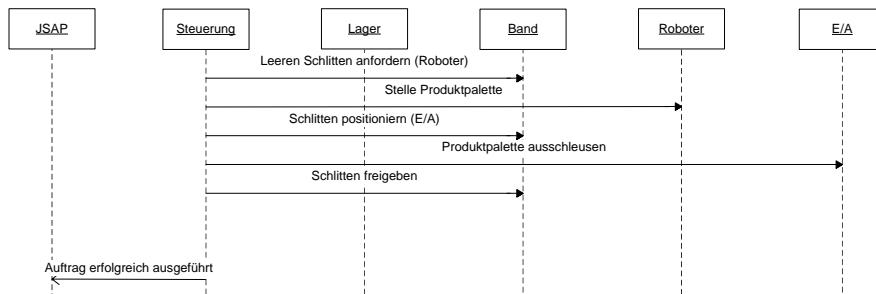


Abbildung 4.7: Queue Produktpalette ausschleusen

Befehl	Subsystem	Parameter	Rückgabe
Leeren Schlitten anfordern(Roboter)	BAND		SchlittenID
Gib Produktpalette	ROBOTER		
Schlitten positionieren(E/A)	BAND	SchlittenID	
Produktpalette ausschleusen	E/A	Matrix	
Schlitten freigeben	BAND	SchlittenID	

Tabelle 4.7: Queue Produktpalette ausschleusen

4.1 Steuerung

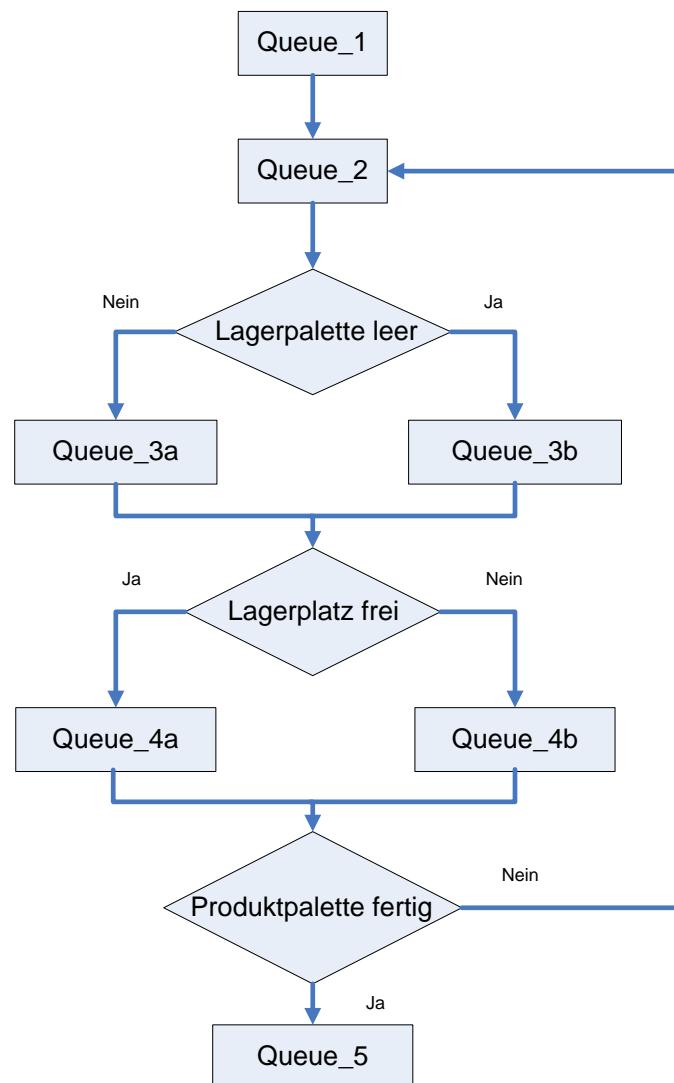


Abbildung 4.8: Ablaufplan Auftrag

4.1 Steuerung

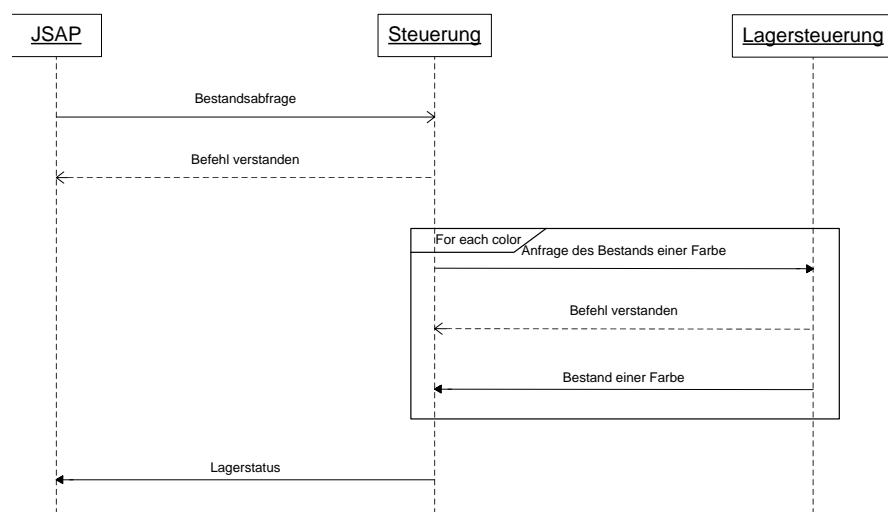


Abbildung 4.9: Bestandsabfrage

4.1 Steuerung

4.1.1.5.2 Aufbau der Bestandsdatei

Die Bestandsdatei hat folgenden Aufbau:

1	<Name des Palettenlagers>
2	[<Farbe><Anzahl>]
3	[<Farbe><Anzahl>]
4	[<Farbe><Anzahl>]
5	[<Farbe><Anzahl>]
6	[<Farbe><Anzahl>]

Tabelle 4.8: Lagerbestand

Zeile 1: Sollten zu einem späteren Zeitpunkt weitere Palettenlager in das System eingebunden werden, muss jedes Lager eindeutig identifiziert werden können. Hierzu dient der Name des Palettenlagers.

Zeilen 2 - 6: In den Zeilen wird zu jeder Farbe die entsprechende Anzahl der sich im Lager befindlichen Smarties gespeichert.

4.1.1.5.3 Beschreibungen der Nicht-Terminalsymbole

<Name des Palettenlagers> : Der Name kann aus einer beliebig langen Zeichenkette bestehen.

<Farbe> : Farbe der Smarties. Erlaubte Werte: r (rot), g (grün), b (blau), y (gelb), w (braun).

<Anzahl> : Anzahl der Smarties einer bestimmten Farbe.

Nachfolgend werden Sequenzdiagramme für den Ablauf einer Steuerung gezeigt und beschrieben. Die Reihenfolge ist dabei so gewählt, dass man einen kompletten Auftrag, der in das System gelangt, bei der Durchführung begleiten kann.

4.1.2 Anmeldung und Verbindungsaufbau der Subsysteme zur Steuerung

Für die Verbindung mit den Subsystemen stellt die Steuerung den Server dar und die Subsysteme agieren als Clients.

4.1 Steuerung

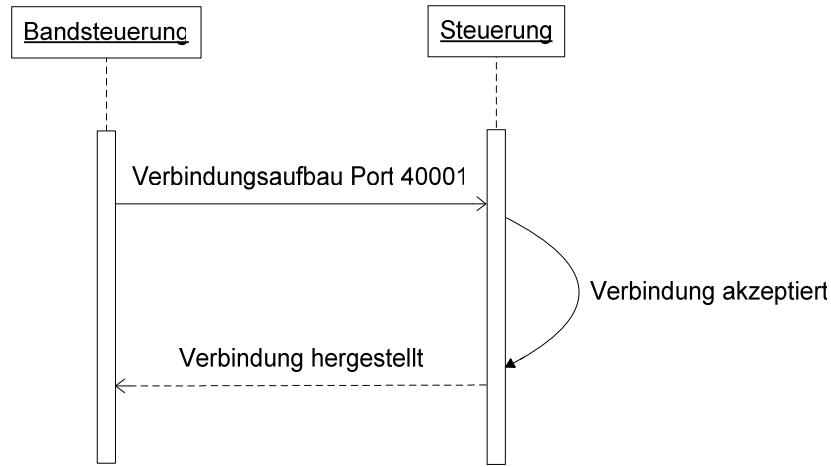


Abbildung 4.10: Beispiel des Verbindungsaufbaus zwischen einer Subsystemsteuerung und der Steuerung

Die Subsysteme melden sich nach dem Hochfahren über verschiedene Ports an der Steuerung an. Jedes Subsystem hat somit seine eigene Verbindung zur Steuerung. Die Grafik ?? zeigt exemplarisch für ein Subsystem die Anmeldung, wie sie im Idealfall abläuft. Folgende Ports wurden für Verbindungen zwischen Steuerung und Subsystemen spezifiziert:

- Port 40001: Verbindung zwischen Steuerung und Bandsteuerung
- Port 40002: Verbindung zwischen Steuerung und Lagersteuerung
- Port 40003: Verbindung zwischen Steuerung und Robotersteuerung
- Port 40004: Verbindung zwischen Steuerung und E/A-Stationssteuerung

Nur wenn sich alle Subsysteme mit der Steuerung verbunden haben kann das System arbeiten und eine Verbindung zwischen Steuerung und JSAP aufbauen, um Aufträge zu erhalten. Sollten sich nicht alle Subsysteme verbinden, so wird nach einem Timeout der Befehl „Herunterfahren“ an die angemeldeten Subsystem geschickt. Für diesen Fehlerfall siehe Kapitel ??.

4.1.3 Anmeldung und Verbindungsaufbau der Steuerung an JSAP

Beim Verbindungsaufbau der Steuerung zum JSAP agiert die Steuerung als Client und JSAP agiert als Server.

4.1 Steuerung

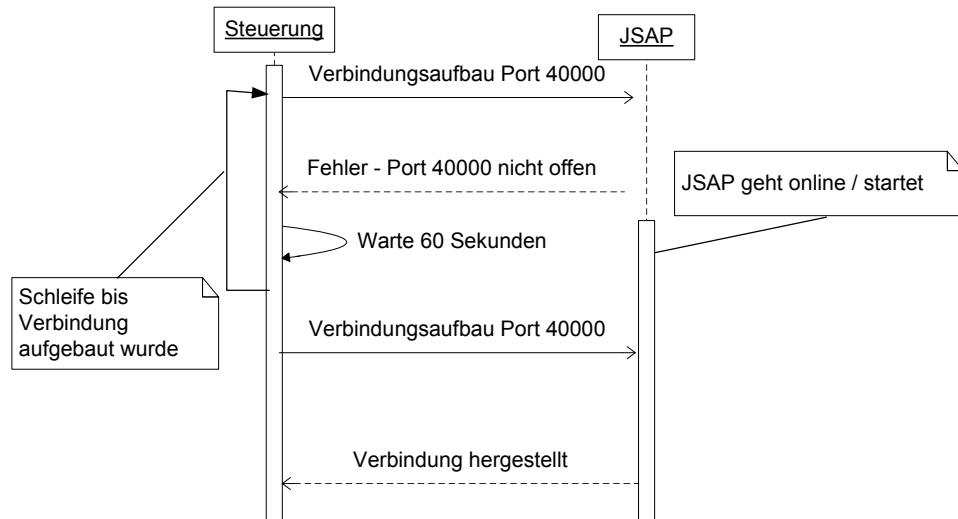


Abbildung 4.11: Anmeldung und Verbindungsauftbau der Steuerung an JSAP

Nachdem alle Subsysteme sich bei der Steuerung angemeldet haben, versucht die Steuerung eine Verbindung zu JSAP herzustellen. Für den Verbindungsauftbau zum JSAP wurde folgender Port spezifiziert:

- Port 40000: Verbindung zwischen Steuerung und JSAP

Sollte dies nicht gelingen, wartet die Steuerung 60 Sekunden, um anschließend wieder zu versuchen, eine Verbindung aufzubauen. Dies wiederholt sich solange, bis eine Verbindung mit JSAP erfolgreich hergestellt werden konnte. Nachdem die Verbindung erfolgreich aufgebaut wurde, wartet die Steuerung auf Befehle von JSAP.

4.1.4 Neuer Auftrag an Steuerung

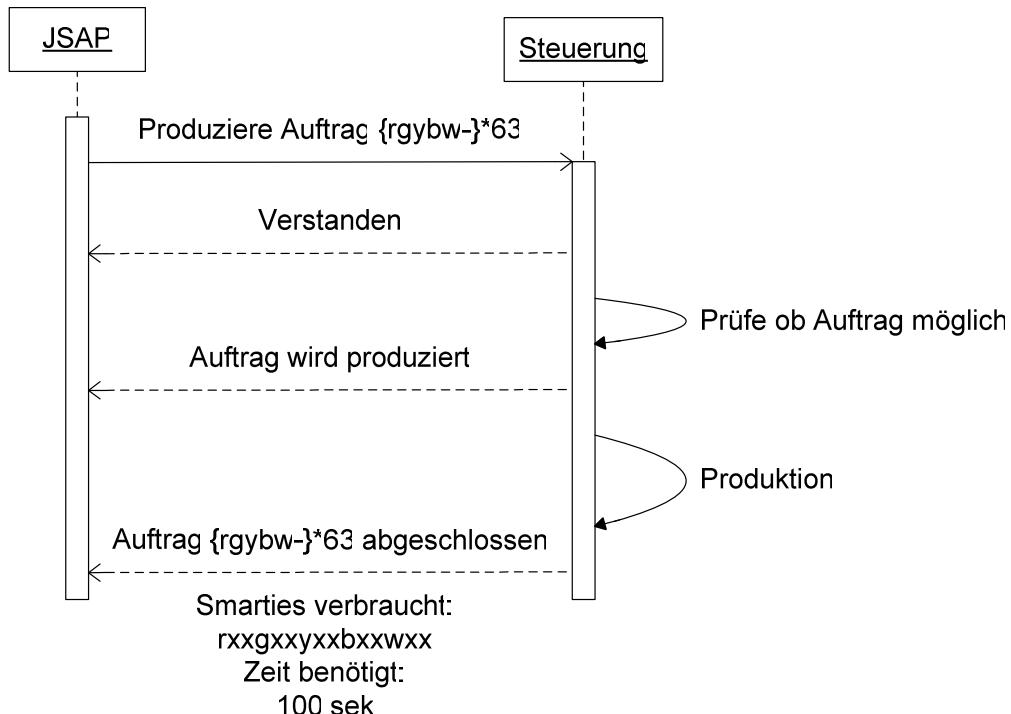


Abbildung 4.12: Neuer Auftrag an Steuerung - Auftrag kann produziert werden

Wird vom JSAP an die Steuerung der Auftrag verschickt und versteht die Steuerung diesen Auftrag, wird die Rückmeldung „Verstanden“ an das JSAP übermittelt. Danach überprüft die Steuerung, ob der Auftrag produziert werden kann. Stellt sie intern fest, dass es möglich ist den Auftrag zu produzieren, dann wird mit der Produktion begonnen und dies JSAP mitgeteilt. Wenn die fertiggestellte Produktpalette an der E/A-Station ausgeschleust wurde, wird JSAP mitgeteilt, dass der Auftrag nun abgeschlossen ist und die Steuerung übermittelt JSAP die Anzahl & Farbe der verwendeten Smarties.

4.1 Steuerung

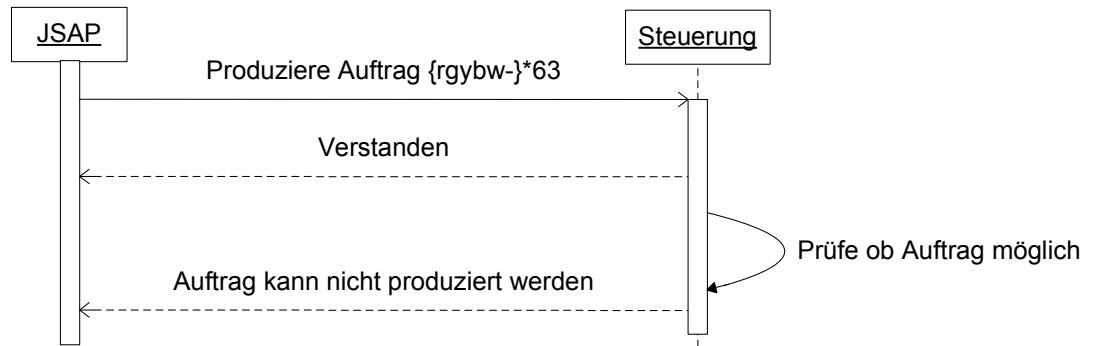


Abbildung 4.13: Neuer Auftrag an Steuerung - Auftrag kann nicht produziert werden

Ist es nicht möglich den Auftrag zu produzieren, weil

- gerade ein Auftrag produziert wird, oder
- im Lager nicht mehr genügend Smarties einer bestimmten Farbe sind,

dann wird dem JSAP mitgeteilt, dass der gewünschte Auftrag nicht produziert werden kann.

4.1 Steuerung

Prüfung eines Auftrages ob Produktion möglich

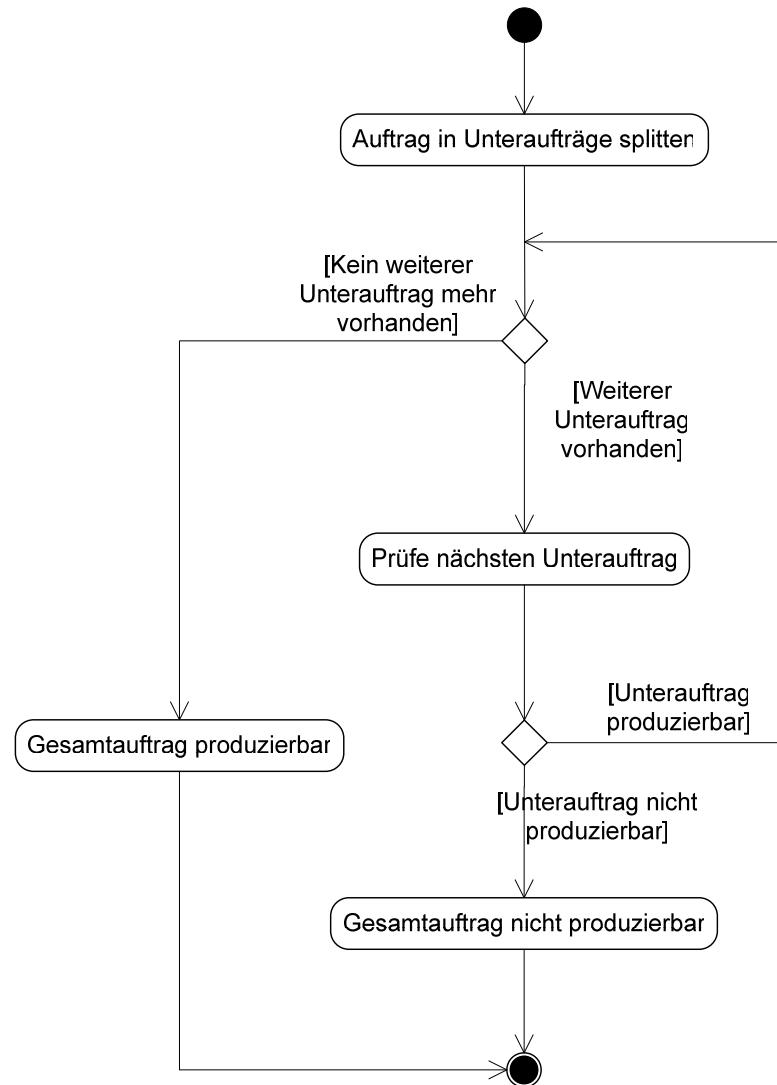


Abbildung 4.14: Prüfung ob Auftrag möglich

Abbildung ?? zeigt, wie die Steuerung prüft ob ein Auftrag produziert werden kann. Dazu wird der Auftrag in einfarbige Unteraufträge aufgeteilt. Auch wenn ein Auftrag nur eine Farbe beinhaltet, wird daraus ein Unterauftrag erstellt. Anschließend wird für jeden Unterauftrag geprüft ob noch genügend Smarties von der Farbe des Unterauftrages im Lager vorhanden sind (Siehe Abbildung ??). Ist der Unterauftrag erfüllbar, dann wird der nächste geprüft. Nach der Prüfung aller Unteraufträge wird der Gesamtauftrag als produzierbar eingestuft und mit dem Produktionsablauf begonnen. Falls für einen Unterauftrag nicht genügend Smarties in der benötigten Farbe vorhanden sind, wird der Gesamtauftrag als nicht produzierbar eingestuft und damit abgelehnt.

4.1 Steuerung

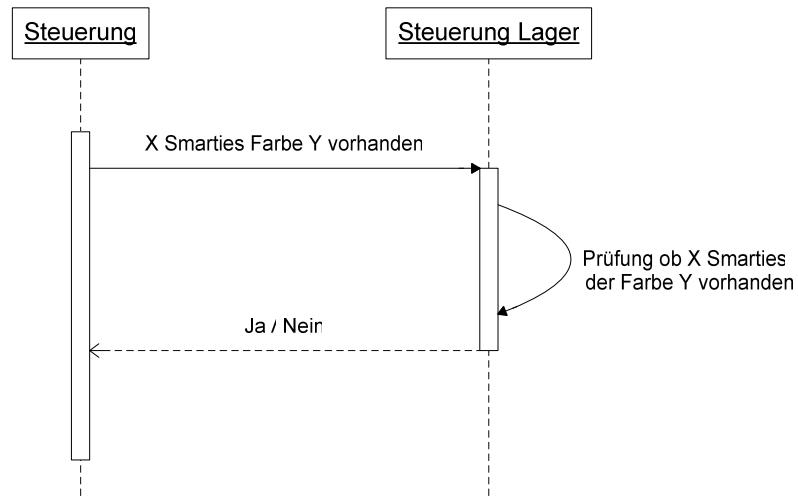


Abbildung 4.15: Prüfung ob genügend Smarties für einen Unterauftrag vorhanden sind

Für jeden Unterauftrag wird eine Anfrage an die Lagersteuerung geschickt, die nachfragt, ob genügend Smarties in der benötigten Farbe vorhanden sind (Siehe Kapitel über Lager ??).

4.1 Steuerung

4.1.5 Einschleusen einer leeren Produktpalette

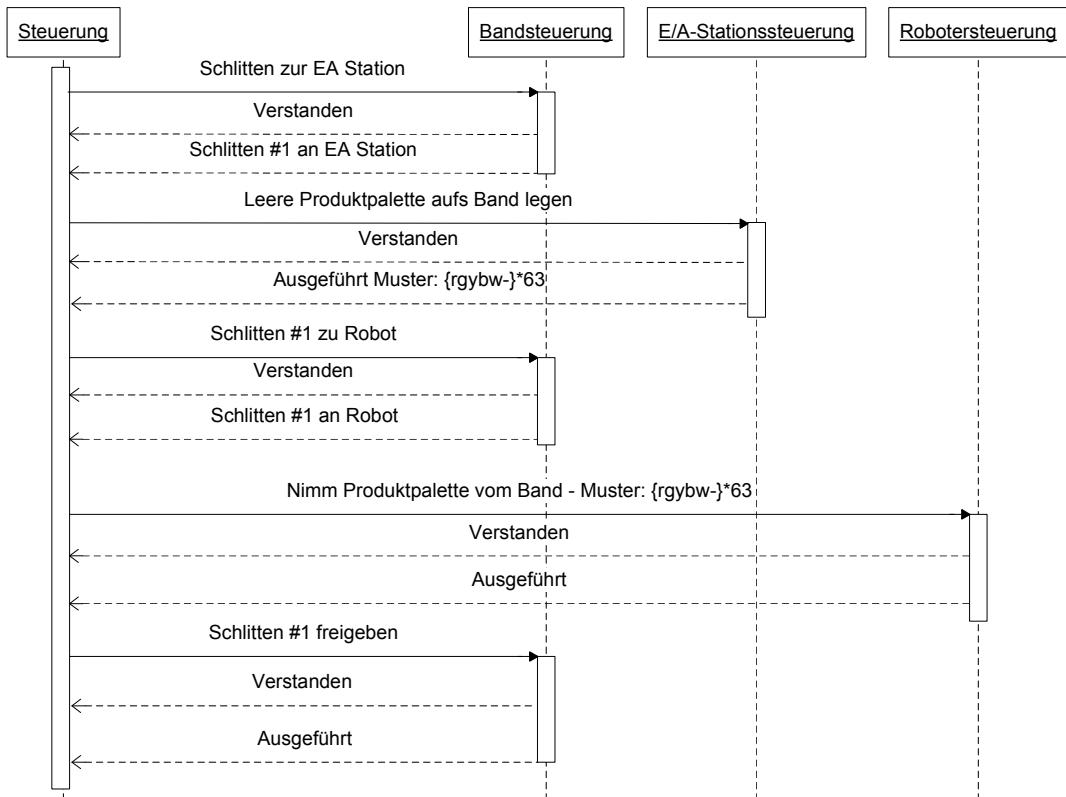


Abbildung 4.16: Einschleusen einer leeren Produktpalette

Da immer nur ein Auftrag abgearbeitet wird, befindet sich zum Beginn eines Zyklus nie eine Produktpalette im System. Diese muss zuerst immer eingeschleust werden. Die Steuerung beauftragt hierzu den Teil „Bandsteuerung“, einen leeren Schlitten vor der E/A-Station zu positionieren. Wurde der Schlitten erfolgreich positioniert, bekommt die Steuerung vom Teil Bandsteuerung eine Schlitten-ID zur weiteren Verwendung zugewiesen. Die E/A-Stationssteuerung erhält dann den Auftrag, eine leere Produktpalette auf den Schlitten zu legen. Da es sich um eine Produktpalette handelt, ist eine Zuordnung Schlitten-ID - Paletten-ID nicht erforderlich. Eine eindeutige Identifikation ist durch die Schlitten-ID gegeben. Ist die leere Produktpalette auf dem Schlitten, muss die Bandsteuerung diesen zum Roboter bringen. Hierzu wird dem Teil Bandsteuerung die Schlitten-ID mitgegeben. Die Bandsteuerung meldet der Steuerung die erfolgreiche Ausführung des Befehls und gibt ihr die Schlitten-ID zurück. Sollte die zurückgegebene Schlitten-ID nicht mit der ursprünglich empfangenen übereinstimmen, muss die Steuerung die Fehlerbehandlung durchführen. Der Teil „Robotersteuerung“ bekommt den Auftrag, die leere Produktpalette vom Band zu nehmen. Sobald die Robotersteuerung die Abnahme quittiert hat, wird der Schlitten für die Bandsteuerung freigegeben. Durch das Attribut Schlitten-ID kann die Bandsteuerung die weitere Verwendung dieses Schlittens koordinieren.

4.1 Steuerung

4.1.6 Lagerpalette von Lager zu Roboter

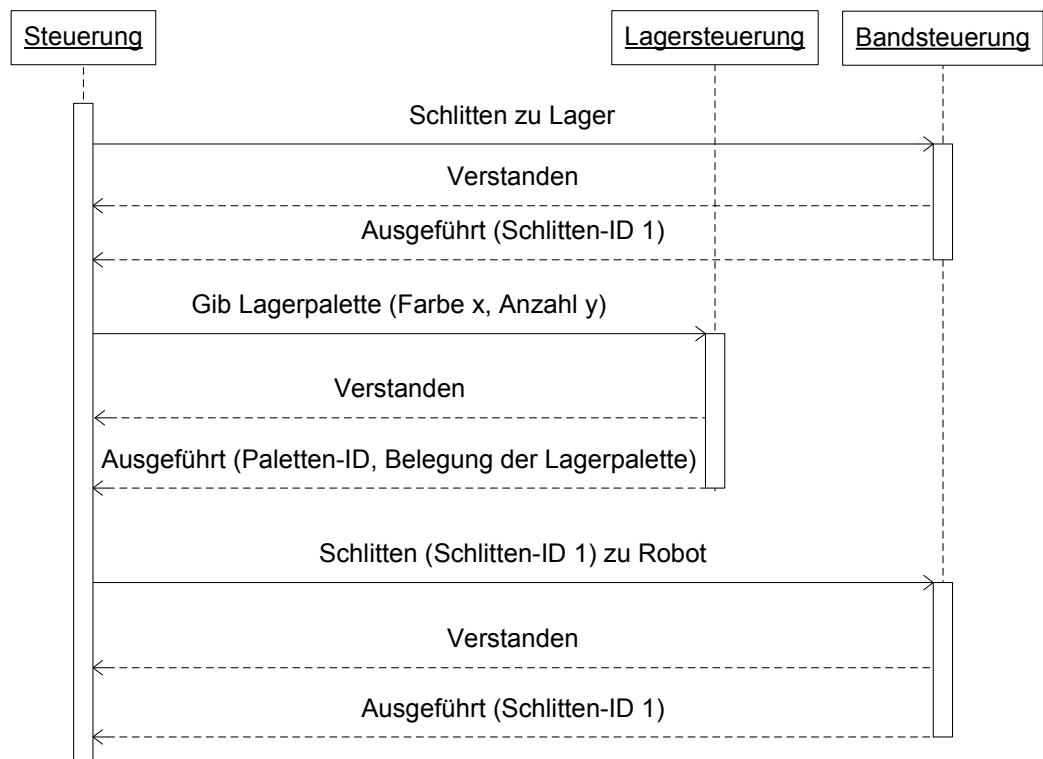


Abbildung 4.17: Befehlsfolge für eine Lagerpalette vom Lager zum Roboter

Die Steuerung fordert von der Bandsteuerung einen Schlitten zum Lager. Wird die Schlitten-ID zurückgegeben, war die Anforderung erfolgreich. Die Lagersteuerung bekommt danach den Auftrag, eine Lagerpalette mit der Farbe X und Anzahl Y auf das Band zu legen. Bei erfolgreicher Anforderung bekommt Steuerung die Paletten-ID und die Belegung der Lagerpalette als Rückgabewerte zurück. Anschließend erhält die Bandsteuerung den Auftrag, den beladenen Schlitten zum Roboter zu fahren. Dabei wird der Bandsteuerung die Schlitten-ID mitgegeben. Bandsteuerung bestätigt die Ausführung des Befehls und gibt die Schlitten-ID an die Steuerung zurück.

4.1 Steuerung

4.1.6.1 Lagerpalette mit geringerer Anzahl als angefordert

Für den Fall, dass das Lager bei der Anforderung einer Lagerpalette mit einer bestimmten Farbe und Anzahl, eine Lagerpalette mit einer geringeren Anzahl an Smarties auslager, muss die Steuerung für den aktuellen Unterauftrag den Bestückungsbefehl ändern und einen weiteren Unterauftrag mit der gleichen Farbe und geänderter Anzahl ans Ende anstellen. Ob die Anzahl der Smarties auf der Lagerpalette für den Unterauftrag reicht, erfährt die Steuerung durch die Meldung „Ausgeführt“ von der Lagersteuerung. In dieser Meldung ist die Paletten-ID und die Belegung der Lagerpalette enthalten.

Daraufhin ändert die Steuerung für den aktuellen Unterauftrag die Smartieanzahl für den Bestückebefehl auf die zurückgegebene Anzahl und erzeugt einen neuen Unterauftrag mit der Differenz zum ursprünglichen Unterauftrag. Dieser wird an das Ende der Unterauftragskette angefügt.

4.1 Steuerung

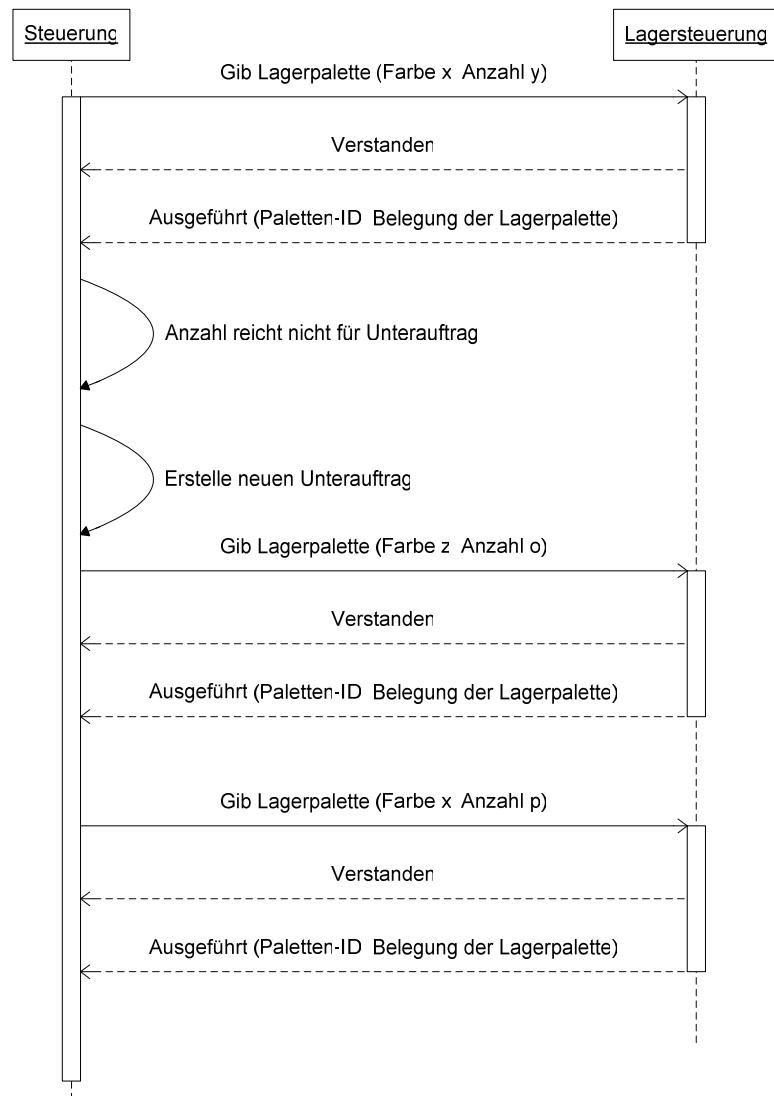


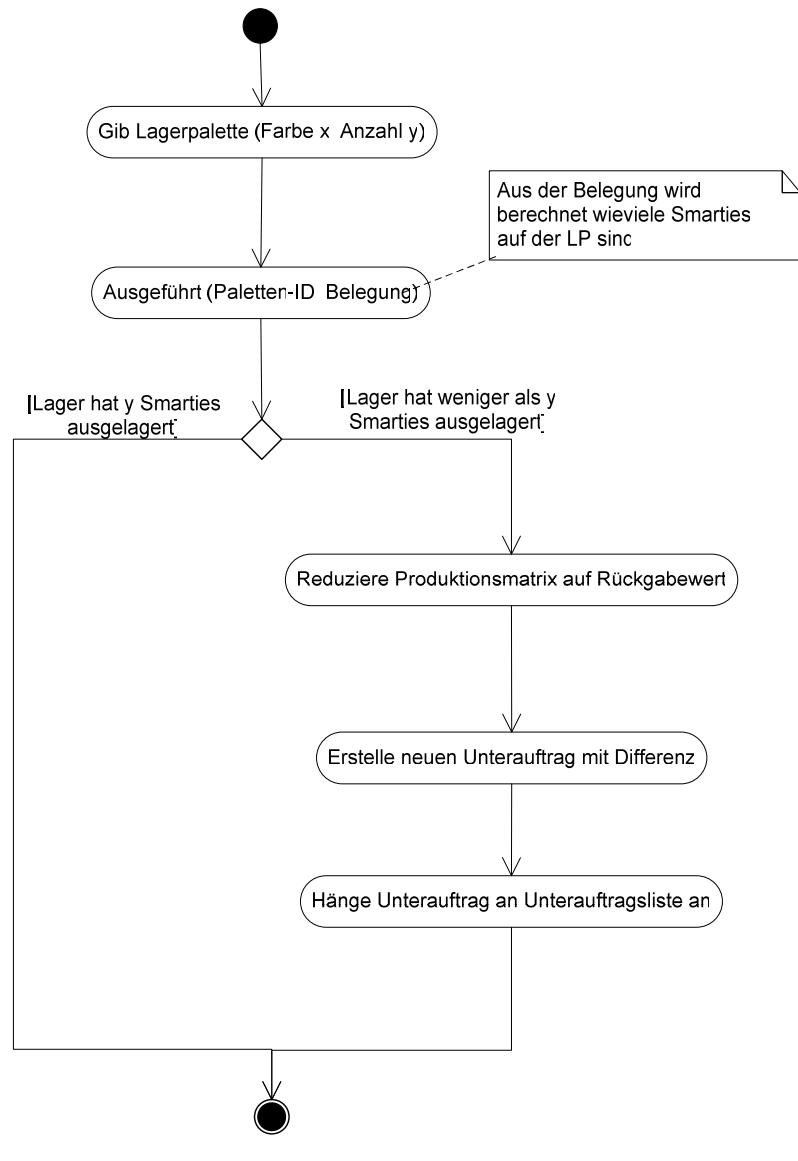
Abbildung 4.18: Sequenzdiagramm für den Ablauf bei zu wenigen Smarties auf der angeforderten Lagerpalette

4.1 Steuerung

Damit hat die Steuerung wieder alle Unteraufträge unter Kontrolle und kann die Produktpalette fertig produzieren. Der Fall, dass für einen Unterauftrag nicht genügend Smarties vorhanden sind, kann an dieser Stelle nicht vorkommen, da bei der Auftragsprüfung von der Lagersteuerung für alle Farben ein OK mitgeteilt wurde (siehe ??).

4.1 Steuerung

Das nachfolgende Ablaufdiagramm zeigt, wie die Steuerung auf das Auslagern einer Palette



reagiert.

Abbildung 4.19: Ablaufdiagramm zur Entscheidung, ob sich genügend Smarties auf der angeforderten Lagerpalette befinden

4.1 Steuerung

4.1.7 Roboter nimmt Lagerpalette vom Band und Schlittenfreigabe

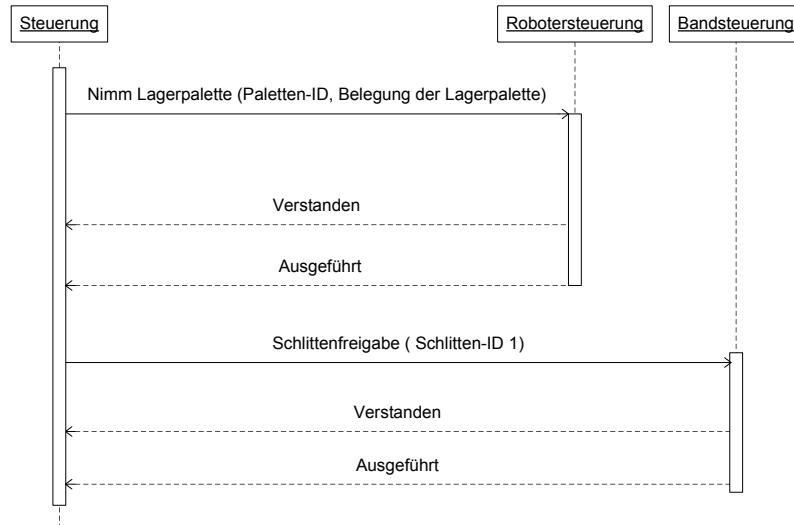


Abbildung 4.20: Befehlsfolge für das Nehmen einer Lagerpalette am Roboter

Dieser Vorgang wird bei der Auftragsbearbeitung im ersten Unterauftrag an die Robotersteuerung geschickt. Steht der Schlitten vor dem Roboter, bekommt die Robotersteuerung den Befehl, die Lagerpalette mit der Farbe X vom Band zu nehmen. Hierbei über gibt die Steuerung der Robotersteuerung die Paletten-ID und die Belegung der Lagerpalette. Nachdem dieser Befehl ausgeführt wurde, informiert die Steuerung die Bandsteuerung, dass der Schlitten freigegeben wird. Steuerung wartet auf „Verstanden“ und „Ausgeführt“.

4.1 Steuerung

4.1.8 Roboter gibt Lagerpalette auf Band

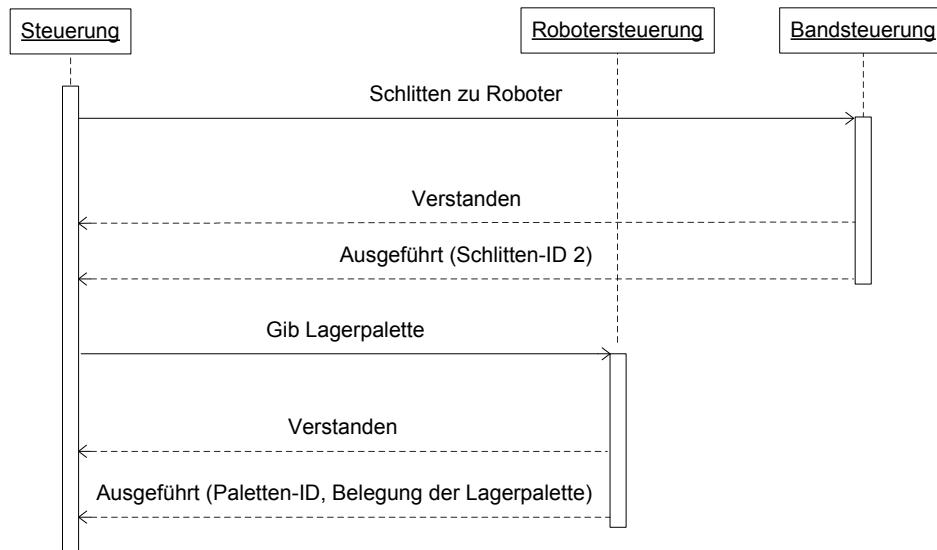


Abbildung 4.21: Befehlsfolge für das Geben einer Lagerpalette am Roboter

Nach dem letzten Unterauftrag wird dieser Befehl aus Abbildung ?? von der Steuerung an die Robotersteuerung geschickt. Dazu fordert die Steuerung von der Bandsteuerung einen Schlitten zum Roboter. Wird die Schlitten-ID zurückgegeben, war die Anforderung erfolgreich. Die Robotersteuerung bekommt danach den Auftrag, die Lagerpalette auf das Band zu legen. Bei erfolgreicher Anforderung bekommt Steuerung die Paletten-ID, die Farb-ID und die Belegung der Lagerpalette als Rückgabewerte zurück. Sollte die Lagerpalette leer sein, so wird die Steuerung wie in Kapitel ?? beschrieben verfahren.

4.1 Steuerung

4.1.9 Tauschebefehl

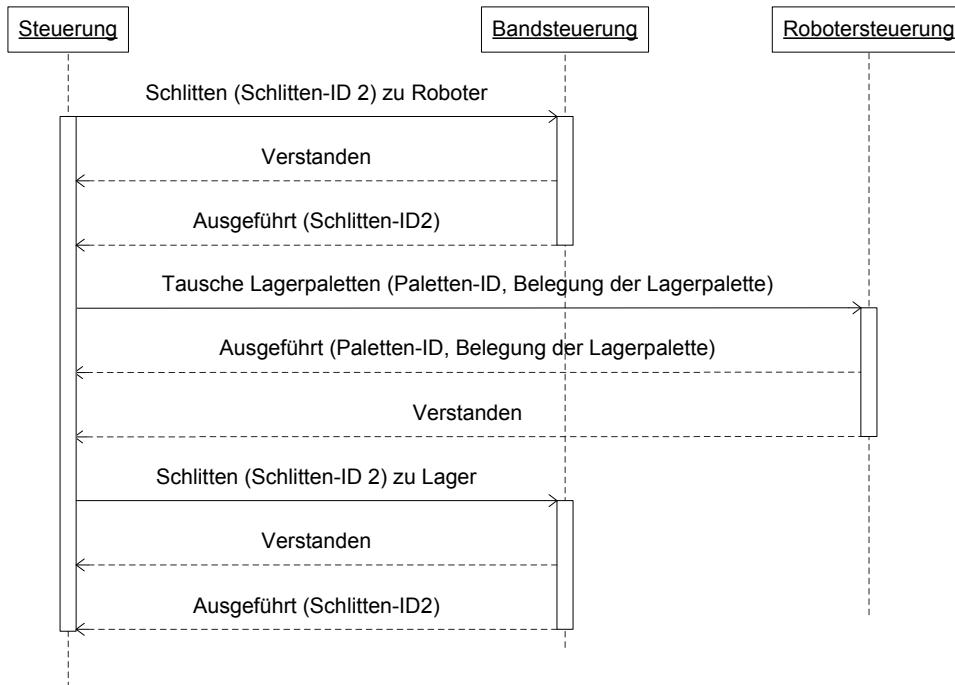


Abbildung 4.22: Befehlsfolge für das Tauschen einer Lagerpalette am Roboter

Steuerung verwendet den Tauschebefehl, wenn bereits eine Lagerpalette beim Roboter liegt und diese mit einer neuen Lagerpalette ausgetauscht werden soll. Dazu bekommt die Robotersteuerung den Befehl „Tausche Lagerpalette“ mit der Paletten-ID und der Belegung der neuen Lagerpalette. Als Rückgabe wird von der Robotersteuerung ebenfalls die Paletten-ID und die Belegung der „alten“ Lagerpalette erwartet. Für eine detaillierte Beschreibung des Vorgangs sei auf Kapitel ?? verwiesen. Solite die vom Roboter auf den Schlitten gelegte Lagerpalette leer sein, so wird diese zur E/A-Station anstatt zum Lager gebracht. Dieser Vorgang ist in Kapitel ?? beschrieben.

4.1 Steuerung

4.1.10 Lagerpalette von Roboter zu Lager, Einschlichten und Schlittenfreigabe

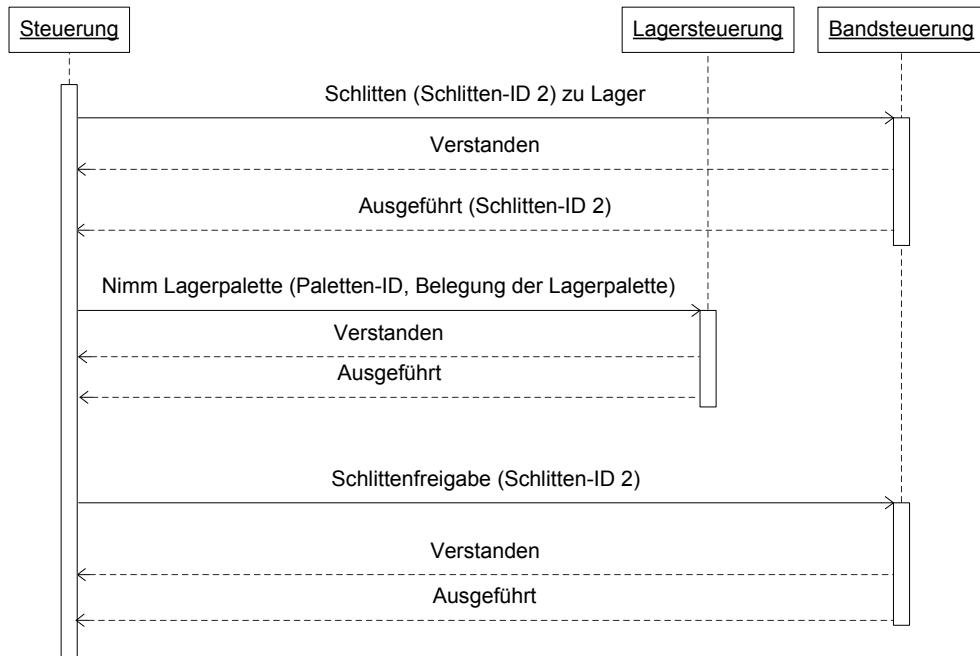


Abbildung 4.23: Befehlsfolge für das Einlagern einer Lagerpalette

Die Bandsteuerung erhält den Auftrag, den beladenen Schlitten zum Lager zu bringen. Dabei wird der Bandsteuerung die Schlitten-ID mitgegeben. Die Bandsteuerung bestätigt die Ausführung des Befehls und gibt die Schlitten-ID an die Steuerung zurück. Steht der Schlitten vor dem Lager, bekommt die Lagersteuerung den Befehl, die Lagerpalette mit der Farbe Y vom Band zu nehmen und einzusortieren. Hierbei übergibt die Steuerung der Lagersteuerung die Farb-ID und die Belegung der Lagerpalette. Nachdem dieser Befehl ausgeführt wurde, informiert die Steuerung die Bandsteuerung, dass der Schlitten freigegeben wird. Steuerung wartet auf „Verstanden“ und „Ausgeführt“. Sollte die Lagerpalette auf dem Schlitten leer sein, so wird diese zur E/A-Station anstatt zum Lager gebracht. Dieser Vorgang ist im Kapitel ?? beschrieben.

4.1.11 Bestückungsbefehl

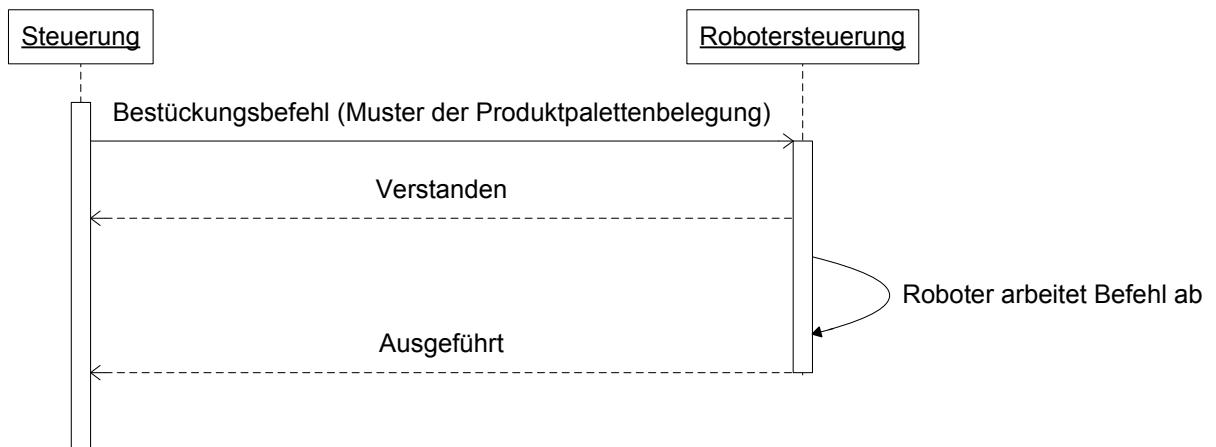


Abbildung 4.24: Befehlsfolge für das Bestücken einer Produktpalette am Roboter

Die Steuerung gibt der Robotersteuerung den Bestückungsbefehl. Der Bestückungsbefehl enthält zudem das Muster für die Belegung der Produktpalette und die Farbe, aus der das Muster besteht. Der Bestückungsbefehl wird dann ausgeführt, wenn eine Produktpalette und eine Lagerpalette auf dem Roboterarbeitsplatz vorhanden sind. Sollte der Roboter mit dem Bestückungsauftrag fertig sein, gibt die Robotersteuerung der Steuerung ein „Ausgeführt“ zurück. Damit kann Steuerung seine Arbeit weiterführen.

4.1 Steuerung

4.1.12 Kombinationsmöglichkeiten aus den Punkten ?? bis ??

Allen Aufträgen ist gleich, dass zu Beginn eines Auftrages immer, wie in ?? beschrieben, eine leere Produktpalette eingeschleust wird und am Ende des Auftrages eine bestückte Produktpalette ausgeschleust wird (Siehe Kapitel ??). Einige Schritte werden allerdings zur Auftragserfüllung mehrfach benötigt. Nachfolgend sind die Kombinationsmöglichkeiten aufgeführt, die die Steuerung für ihre Auftragserledigung hat. Dabei wird davon ausgegangen, dass der Auftrag aus mehreren Farben besteht und zur Abarbeitung eines Unterauftrages sich immer genügend Smarties auf der Lagerpalette befinden.

- Eine Lagerpalette wird aus dem Lager geholt und an eine Lagerposition von Roboter gebracht:
 - ?? Lagerpalette von Lager zu Roboter
 - ?? Roboter nimmt Lagerpalette vom Band und Schlittenfreigabe
- Eine Lagerpalette aus dem Lager wird mit einer Lagerpalette an einer Lagerposition von Roboter getauscht:
 - ?? Lagerpalette von Lager zu Roboter
 - ?? Tauschbefehl
 - ?? Lagerpalette von Roboter zu Lager, Einsortieren und Schlittenfreigabe
- Eine Lagerpalette an einer Lagerposition von Roboter wird an das Lager geschickt und dort eingelagert:
 - ?? Roboter gibt Lagerpalette auf Band
 - ?? Lagerpalette von Roboter zu Lager, Einschlichten und Schlittenfreigabe

Wie schon erwähnt, wird der Bestückungsbefehl immer dann aufgerufen, wenn eine Produktpalette sich an der Roboterstation befindet und eine Lagerpalette gerade auf den Roboterarbeitsplatz abgelegt wurde.

4.1 Steuerung

4.1.13 Produktpalette ist fertig und wird ausgeschleust

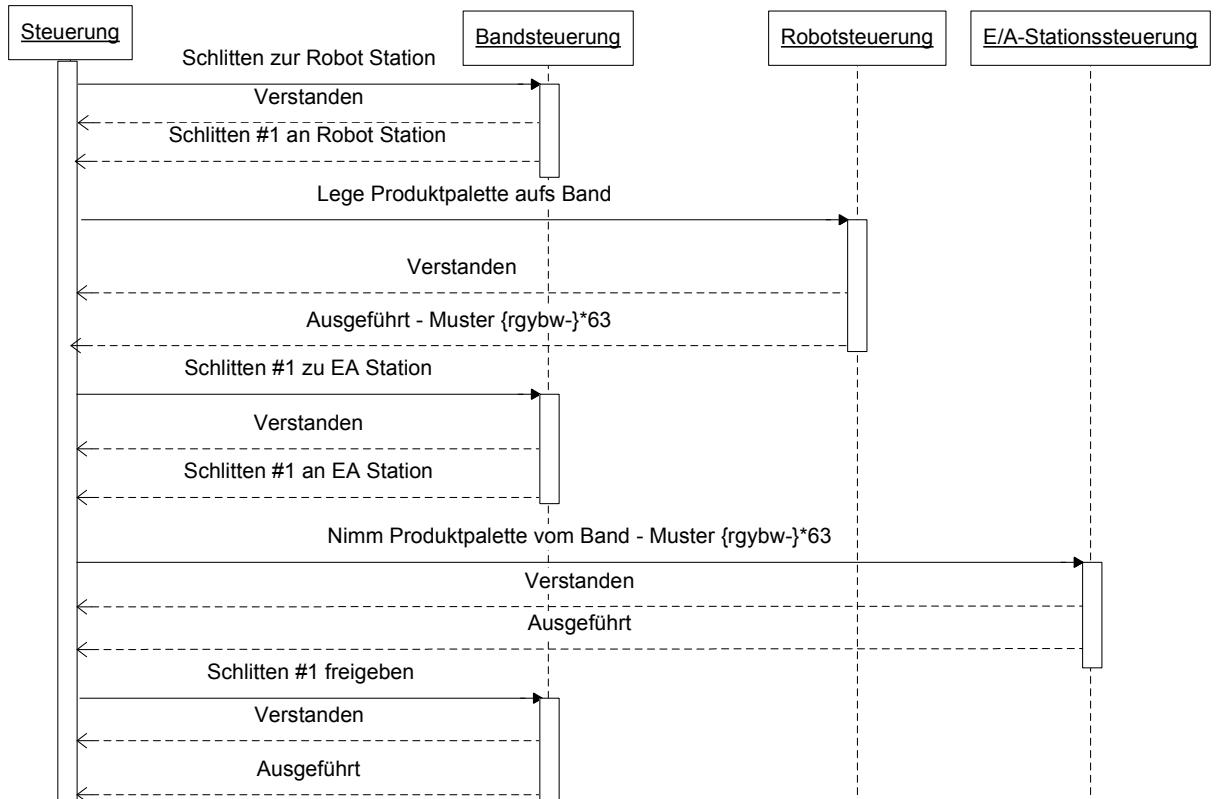


Abbildung 4.25: Fertig bestückte Produktpalette von Roboter zur E/A-Station und ausschleußen

Stellt die Steuerung fest, dass die Produktpalette fertig bestückt wurde, dann ist der nächste Arbeitsabschnitt, diese vom Roboter abzuholen und an der E/A-Station auszuschleusen. Das Sequenzdiagramm (Abbildung ??) zeigt grafisch den Ablauf. Nachdem die Bandsteuerung den Schlitten zum Roboter gebracht hat, wird der Robotersteuerung mitgeteilt, dass nun die Produktpalette auf das Band gestellt werden soll. Wenn dies erfolgt ist, wird der Schlitten zur E/A-Station geschickt. Dort angekommen, wird die E/A-Stationssteuerung informiert, dass die Produktpalette vom Band genommen werden soll. Nachdem die Produktpalette vom Band genommen wurde, wird der Bandsteuerung mitgeteilt, dass der genutzte Schlitten wieder frei ist.

4.1 Steuerung

4.1.14 Leere Lagerpalette vom Roboter zur E/A-Station

Eine leere Lagerpalette wird vom Roboter direkt zur E/A-Station geschickt. Dies ist notwendig, damit die leere Lagerpalette wieder gefüllt werden kann. Das Befüllen ist in Kapitel ?? beschrieben. Dieses Kapitel beschreibt den Ablauf, wie die leere Lagerpalette zur E/A-Station gelangt an Hand von zwei Grafiken.

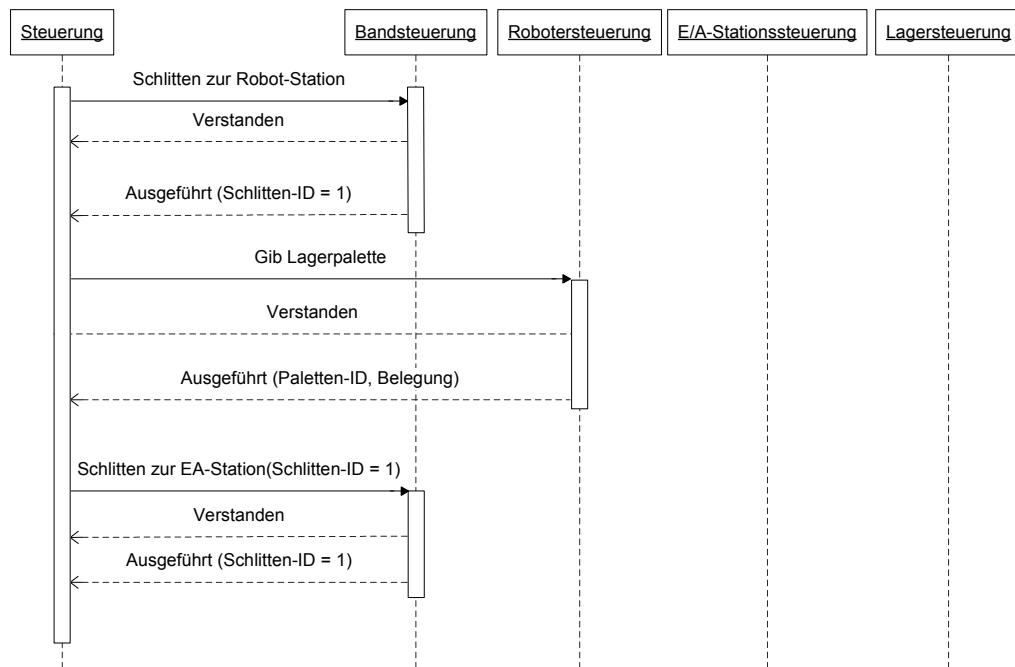


Abbildung 4.26: Befehlsfolge für den Rückweg einer Lagerpalette vom Roboter

Die Steuerung fordert von der Bandsteuerung einen Schlitten zum Roboter. War die Anforderung erfolgreich, wird eine Schlitten-ID an die Steuerung zurück gegeben. Die Robotersteuerung bekommt dann den Auftrag, die Lagerpalette X auf das Band zu legen. Als Rückgabewert wird die Paletten-ID und die Belegung erwartet, auch wenn die Palette leer ist. Anschließend erhält die Bandsteuerung den Auftrag, den beladenen Schlitten zur E/A-Station zu bringen. Da die Beladung des Schlittens für die Bandsteuerung uninteressant ist, wird nur die Schlitten-ID mitgegeben. Die Bandsteuerung bestätigt die Ausführung des Befehls und gibt die Schlitten-ID des beförderten Schlittens zurück. Somit ist für die Steuerung ein eventueller Fehler (z.B. wenn der falsche Schlitten transportiert wurde) erkennbar.

4.1 Steuerung

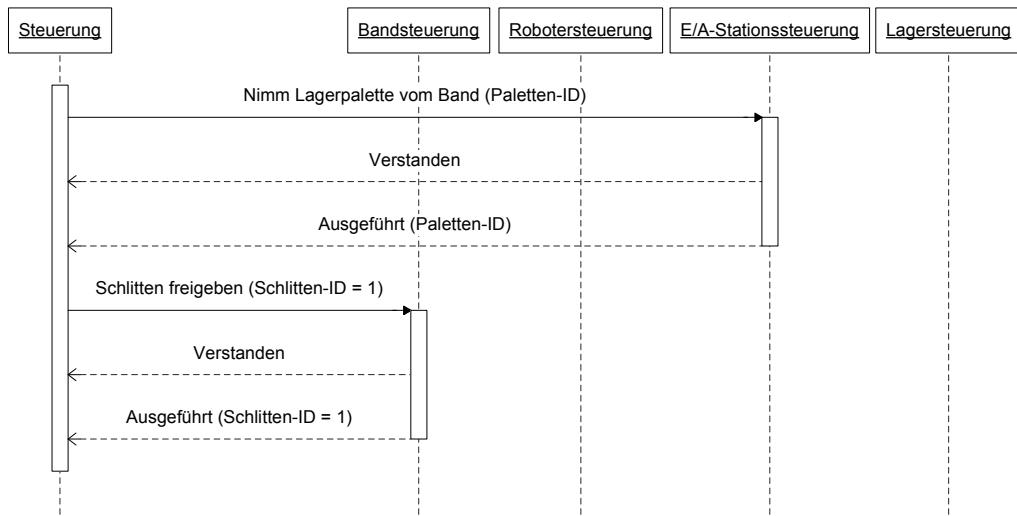


Abbildung 4.27: Befehlsfolge für den Rückweg einer Lagerpalette vom Roboter

Steht der Schlitten vor der E/A-Station, bekommt die E/A-Stationssteuerung den Auftrag, die Lagerpalette mit der Paletten-ID X vom Band zu nehmen. Eine Freigabe des Schlittens für die Bandsteuerung erfolgt nach erfolgreicher Ausführung des Befehls. Die Entscheidung, ob der Schlitten vor der E/A-Station stehen bleibt für eine neue Beladung oder abgezogen wird, liegt nicht in der Verantwortung der Steuerung. Die Steuerung merkt sich, welche Lagerpaletten mit welcher Paletten-ID ausgeschleust wurden, um diese später wieder anzufordern von der E/A-Stationssteuerung. Wenn die leere Lagerpalette ausgeschleust wurde, arbeitet die Steuerung den nächsten Unterauftrag ab. Sind alle Unteraufträge abgearbeitet, wird die fertige Produktpalette ausgeschleust und die Steuerung fährt mit dem Befüllen, wie in Kapitel ?? beschrieben, der leeren Lagerpalette(n) fort.

4.1 Steuerung

4.1.15 Befüllen einer leeren Lagerpalette

Sind alle Unteraufträge abgearbeitet und ist die fertige Produktpalette ausgeschleust worden, so erhält die E/A-Stationssteuerung den Auftrag, die vom Band genommene(n) leere(n) Lagerpalette(n) zu bestücken.

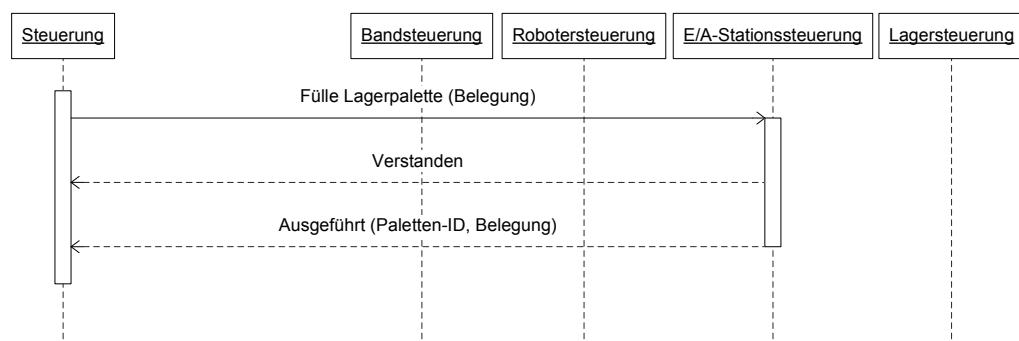


Abbildung 4.28: Befüllbefehl von Steuerung an E/A-Stationssteuerung

Um die leere(n) Lagerpalette(n) zu bestücken, wird die gewünschte Belegung beim Befüllbefehl von der Steuerung an die E/A-Stationssteuerung übergeben.

4.1 Steuerung

4.1.16 Befüllte Lagerpalette einschleusen und einlagern

Anschließend wird die Bandsteuerung beauftragt, einen Schlitten bei der E/A-Station zu positionieren. Entweder steht der Schlitten noch da oder ein neuer Schlitten wird positioniert. Als Rückgabewert wird von der Bandsteuerung in jedem Fall eine Schlitten-ID erwartet, auch wenn der Schlitten an der E/A-Station verblieben ist. Die E/A-Stationssteuerung wird aufgefordert, die befüllte Lagerpalette mit der Paletten-ID Y auf das Band zu legen. Dabei kann die Belegung der Lagerpalette mit der Paletten-ID Y von der angeforderten Belegung abweichen. Die Lagerpalette wird trotzdem angenommen und zum Lager transportiert. Die Farbe der Belegung ist durch einen Kleinbuchstaben spezifiziert. Die Bandsteuerung wird nun beauftragt, den beladenen Schlitten mit der ID z von der E/A-Station zum Lager zu fahren. Die Bandsteuerung bestätigt, wenn der Schlitten dort angekommen ist und gibt die Schlitten-ID zurück.

4.1 Steuerung

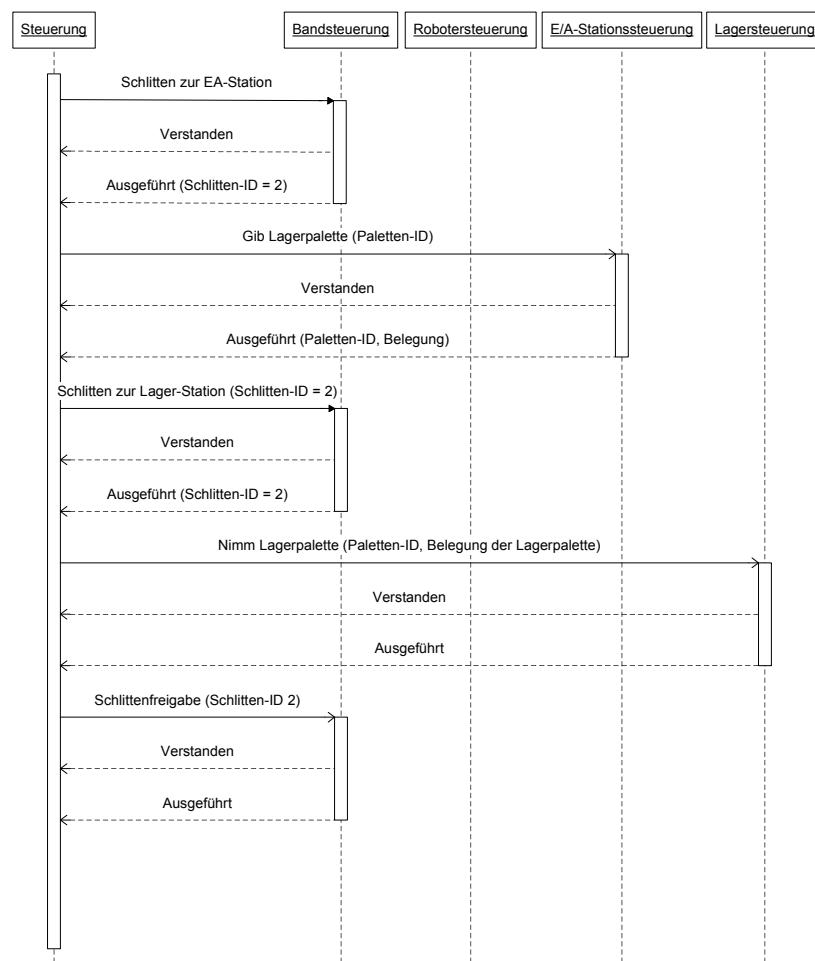


Abbildung 4.29: Befüllte Lagerpalette einschleusen und einlagern

Die Lagersteuerung muss nun die Palette vom Band nehmen, als Attribute werden die Paletten-ID und die Belegung der Lagerpalette mitgegeben. Die weitere Verantwortung, an welcher Position die Palette im Lager platziert wird, liegt bei der Lagersteuerung und ist für die Steuerung uninteressant. Wurde die Lagerpalette vom Band genommen, wird der Schlitten für die Bandsteuerung freigegeben. Als Attribut wird die Schlitten-ID mitgegeben. Mit der Freigabe interessiert die Steuerung der Schlitten nicht mehr, die weitere Verantwortung liegt bei der Bandsteuerung.

4.1.17 Auftrag abbrechen

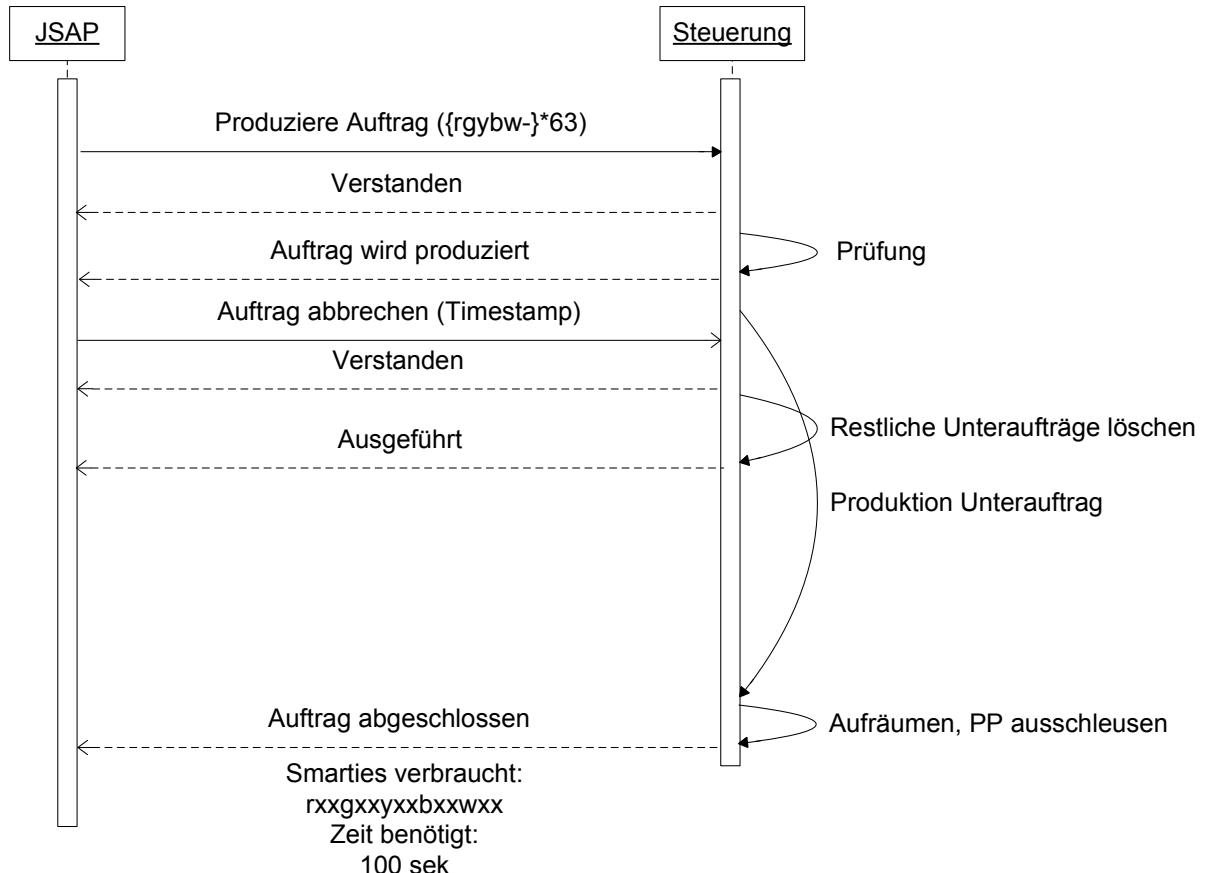


Abbildung 4.30: Auftrag abbrechen

Der gesamte Produktionsverlauf in diesem Dokument ist sequenziell spezifiziert. Es wird daher immer ein Befehl an ein Subsystem geschickt und auf die Abarbeitung gewartet. Der einzige Sonderfall ist „Auftrag abbrechen“. Während ein Auftrag produziert wird, kann JSAP den Auftrag abbrechen. In diesem Fall wird allerdings nicht sofort mit der Produktion aufgehört, sondern erst nachdem der in Produktion befindliche Unterauftrag abgearbeitet wurde. Abbildung ?? stellt gekürzt dar, wie ein Auftrag abgebrochen wird. Um den parallelen Ablauf zu verdeutlichen, wird mit dem Produktionsauftrag (Produziere Auftrag (rgyb-w-*63)) begonnen. Nachdem die Steuerung, wie in Kapitel ?? beschrieben, den Auftrag geprüft und mit der Produktion der Unteraufträge begonnen hat, sendet JSAP einen „Auftrag abbrechen“ Befehl. Als Parameter wird der Timestamp des „Produziere Auftrag“ Befehls mitgeschickt. Die Steuerung löscht die noch nicht bearbeiteten Unteraufträge und lässt währenddessen den aktuell in Bearbeitung befindlichen Unterauftrag normal abschließen. Nach dem Löschen der Unteraufträge wird der Befehl „Auftrag abbrechen“ ausgeführt. Nun folgt, wie in Kapitel ??, das Einlagern aller im System befindlichen Lagerpaletten im Lager oder das Ausschleusen einer leeren Lagerpalette, wie in Kapitel ?? beschrieben. Anschließend muss noch die Produktpalette ausgeschleust werden. Dieser Ablauf ist der gleiche Ablauf, wie wenn ein Auftrag nicht abgebrochen wurde (Siehe Kapitel ??).

4.1.18 Lagerstatus abrufen

JSAP benötigt die Möglichkeit, die Anzahl (pro Farbe) der Smarties in der FDZ-Zelle auslesen zu können. Dazu wird von JSAP eine Anfrage an die Steuerung geschickt. Nach Erhalt der Anfrage wird diese mit „Verstanden“ bestätigt.

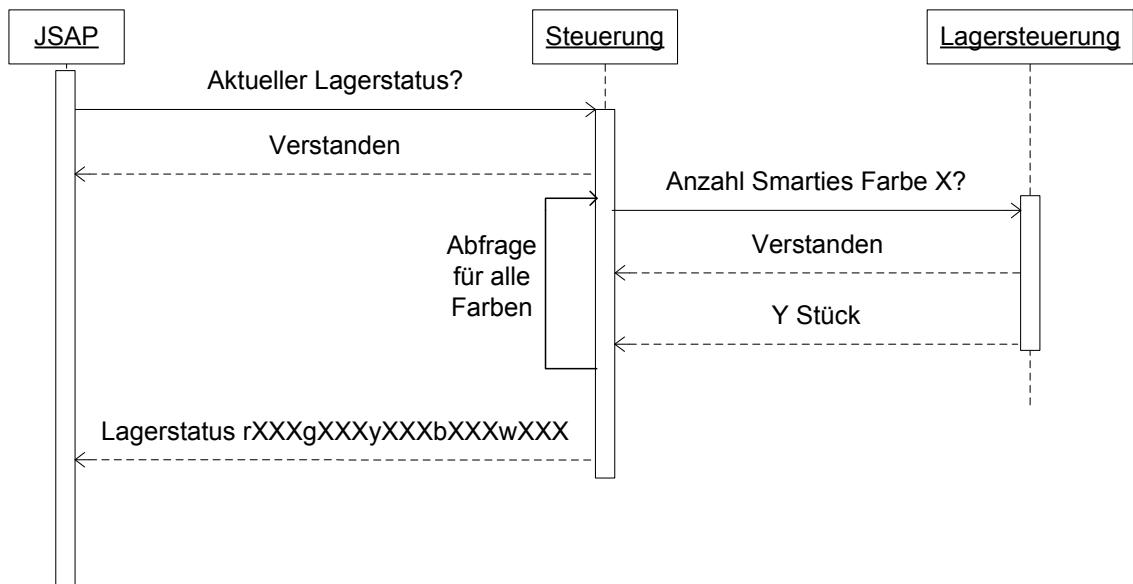


Abbildung 4.31: Lagerstatus abrufen

Wie in Abbildung ?? dargestellt, ruft die Steuerung in Einzelbefehlen für jede vorhandene Farbe die Anzahl der Smarties bei der Lagersteuerung ab. Nachdem die Steuerung sämtliche Daten gesammelt hat, teilt sie JSAP die Anzahl der vorhanden Smarties nach Farben aufgeteilt mit.

Dieser Befehl kann nur ausgeführt werden, wenn Steuerung keinen Auftrag besitzt. In diesem Fall sind alle im System vorhandenen Smarties im Lager eingelagert und können dort zentral abgefragt werden. Eine Abfrage während ein Auftrag produziert wird, ist für diese Version nicht vorgesehen.

4.2 Roboter

4.2 Roboter

4.2.1 Hochfahren/Initialisierung

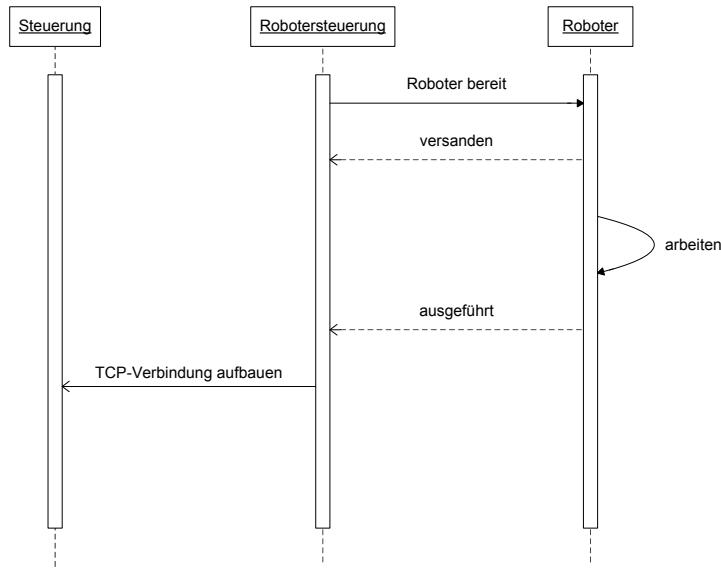


Abbildung 4.32: Initialisierung / Systemstart

Nach Systemstart überprüft die Robotersteuerung (Sr) ob der Roboter (ro) betriebsbereit ist und baut die TCP Verbindungen zu der Steuerung (ST) und dem Roboter (ro) zum Port 30000 auf. Die Robotersteuerung (Sr) ist Server für den Roboter und Client für Steuerung. Zudem wird festgestellt, ob die letzte Aktion erfolgreich ausgeführt wurde (Auftragsspeicherung) und ggf. wird der letzte Befehl beendet.

4.2 Roboter

4.2.2 Lagerpalette (LP) vom Band nehmen (nach A/B)

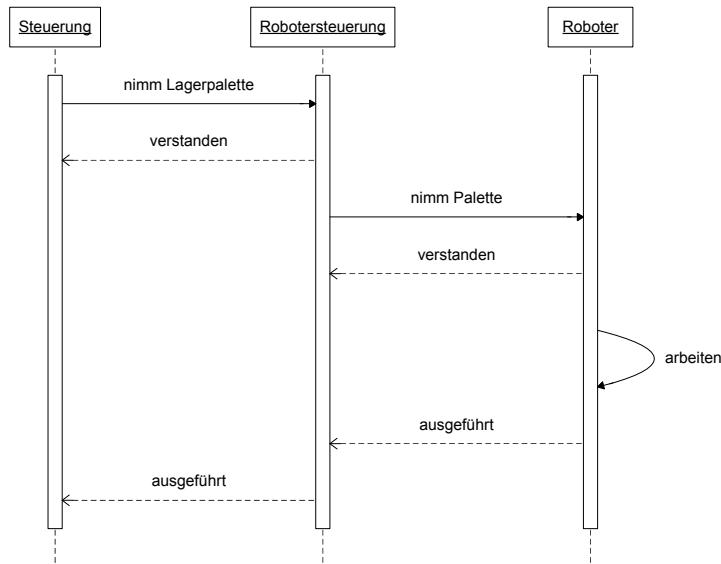


Abbildung 4.33: nimm Lagerpalette

Die Steuerung (ST) sendet an die Robotersteuerung (Sr) den Befehl zum Entfernen einer LP vom Transportband (T). Sr antwortet ST, dass der Befehl verstanden wurde. Sr sendet nun an ro den Elementarbefehl die LP vom Band auf die Position A oder B zu legen. ro sendet seinerseits an Sr, ob der Elementarbefehl verstanden wurde, und abschliessend den Status der ausgeführten Aktion. Sr sendet schliesslich an ST ebenfalls eine Antwort, die das Ergebnis der ausgeführten Aktion wiedergibt. Sr ist jetzt entweder bereit neue Befehle zu empfangen oder befindet sich in einem Fehlerzustand, der die weitere Abarbeitung von Befehlen verhindert.

HINWEIS:

Für die Lagerung und Platzierung der Paletten auf der Arbeitsfläche ist nur die Robotersteuerung verantwortlich. ST muss darüber nichts wissen und sich nicht darum kümmern.

4.2 Roboter

4.2.3 Produktpalette (PP) vom Band nehmen (nach X)

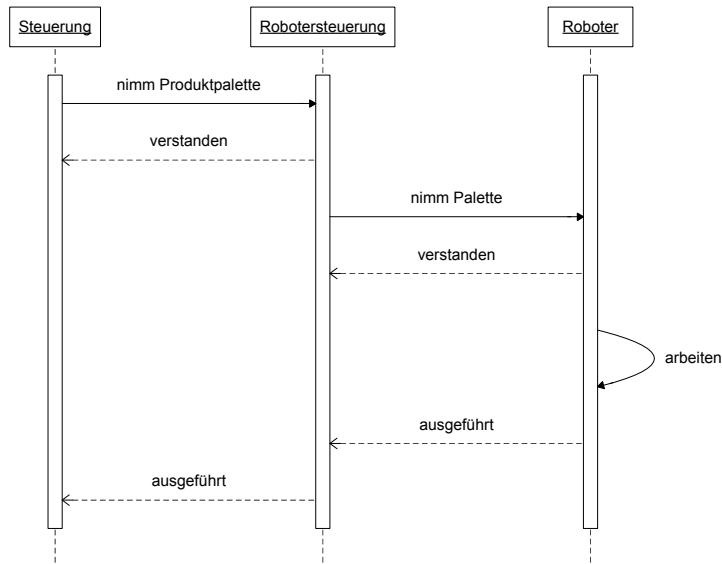


Abbildung 4.34: nimm Produktpalette

Dieser Befehl funktioniert analog zu ??, allerdings wird hier eine PP vom Transportband (T) auf die Arbeitsfläche X gelegt.

4.2 Roboter

4.2.4 PP bestücken

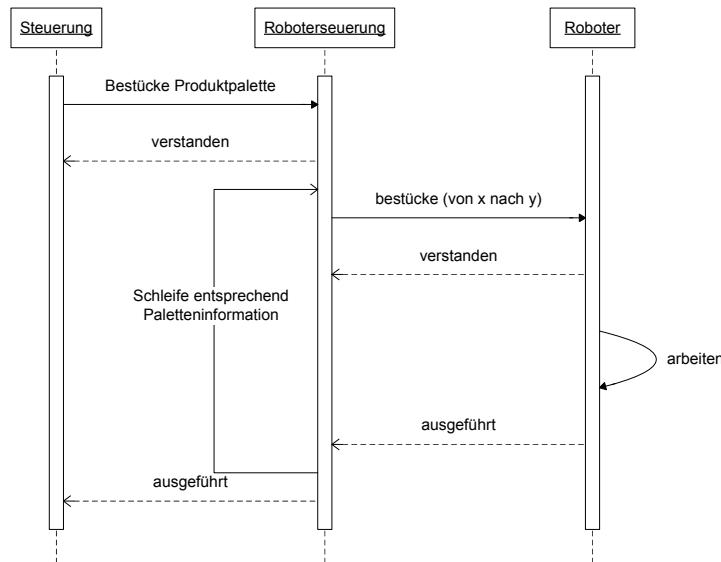


Abbildung 4.35: bestücken

ST sendet an Sr den Bestückungsbefehl, der als Übergabeparameter die Paletteninformation der zu bestückenden PP enthält. Sr antwortet ST ob der Befehl verstanden wurde.

Anschliessend wird in einer Schleife die übergebene Matrix aufgeteilt und pro Smartie mit Ziel- und Quellmatrix an ro gesendet. Für jeden Elementarbefehl an ro wird die übliche Antwortstruktur zwischen Sr und ro angewendet (verstanden und ausgeführt).

Nach Abarbeitung der Matrix wird entweder ein Fehler an ST oder die Antwort "Befehl ausgeführt", inklusive der abgearbeiteten Paletteninformation, gesendet.

Die Paletteninformation der PP vor und nach der Bestückung könnte so aussehen:

Fall 1: LP auf A hat 50 r-Smarties, PP auf X benötigt 10.

vorher{rrrrrrrrrr-----}
danach{-----}

Es wurden alle zu bestückenden Stellen in der Matrix erfolgreich bestückt.

Fall 2: LP auf A hat 16 y-Smarties, PP auf X benötigt 22.

vorher{yyyyyyyyyyyyyyyy-----}
danach{-----XXXXXX-----}

Es konnten nicht alle geforderten Stellen in der Matrix bestückt werden. Dies wird ausgedrückt durch ein "X".

Mehrfachbestückung einer Produktpalette:

LP auf A hat 16 y-Smarties, PP auf X benötigt 3 gelbe Smarties in der ersten Reihe. Nach der ersten Bestückung erfolgt für die selbe Produktpalette eine erneute Bestückung, mit welcher die ersten 5 Positionen besetzt werden sollen.

4.2 Roboter

Die in der Antwort auf die beiden Bestückungsbefehle enthaltene Produktpalettenmatrizen sehen wie folgt aus:

```
erster{-----}  
zweiter{yyy-----}
```

Dabei ist die erste Antwort ein A002, die zweite Antwort der Fehler F006. Die in F006 enthaltenen Farbwerte sind die Farben der Smarties, welche sich bereits auf der PP befinden und mit dem neuen Bestückungsauftrag in Konflikt stehen.

ACHTUNG!

Der Elementarbefehl von Sr an ro zum Bestücken einer Palette ist absolut modular gehalten: der Roboter soll als Übergabewerte eine BELIEBIGE Quellpalette und eine BELIEBIGE Zielpalette erhalten. Es obliegt der Robotersteuerung zu wissen, ob der übergebene Index der Ziel - und Quellpaletten überhaupt existiert. Zur Erinnerung, LP können 91, PP 63 Smarties aufnehmen

4.2.5 PP auf das Band legen (von X)

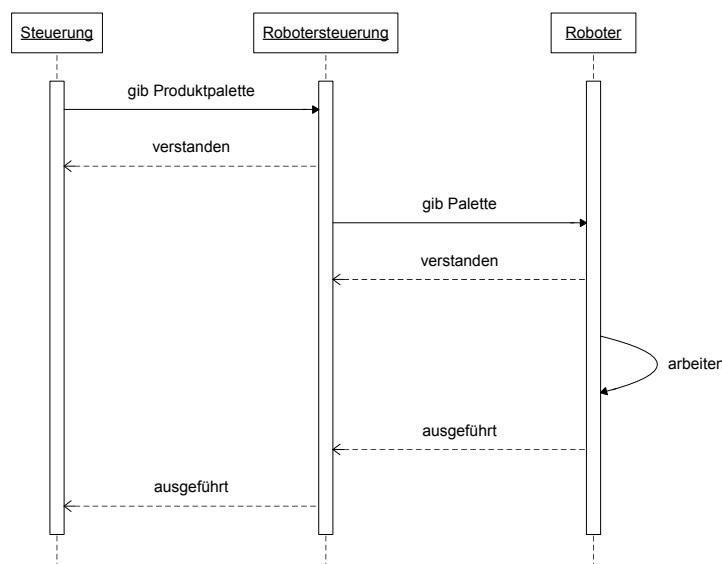


Abbildung 4.36: gib Produktpalette

ST sendet den Befehl an Sr. Der Ablauf dieses Befehls ist analog zu 2., mit dem Unterschied das hier die PP zurück auf T gelegt wird.

4.2 Roboter

4.2.6 LP auf das Band legen/Arbeitsbereich leeren (von A/B)

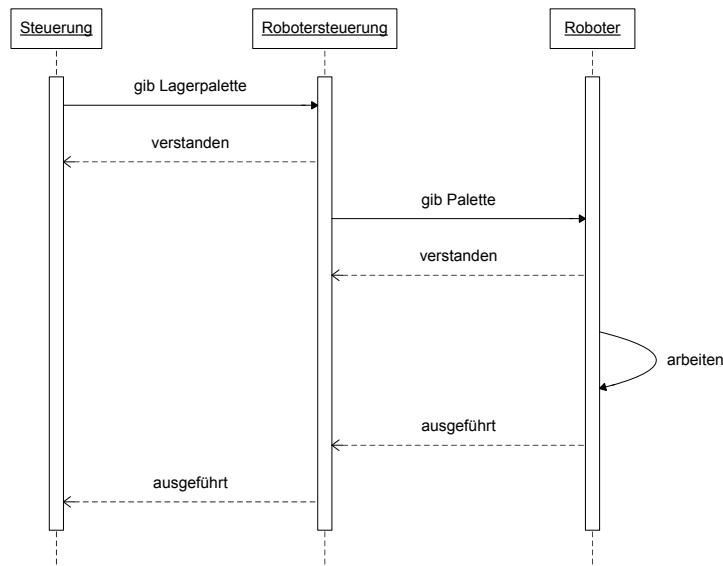


Abbildung 4.37: gib Lagerpalette

Siehe ???. Hier wird allerdings eine LP bewegt.

4.2.7 Tausche LP (von T nach A/B und von A/B nach T)

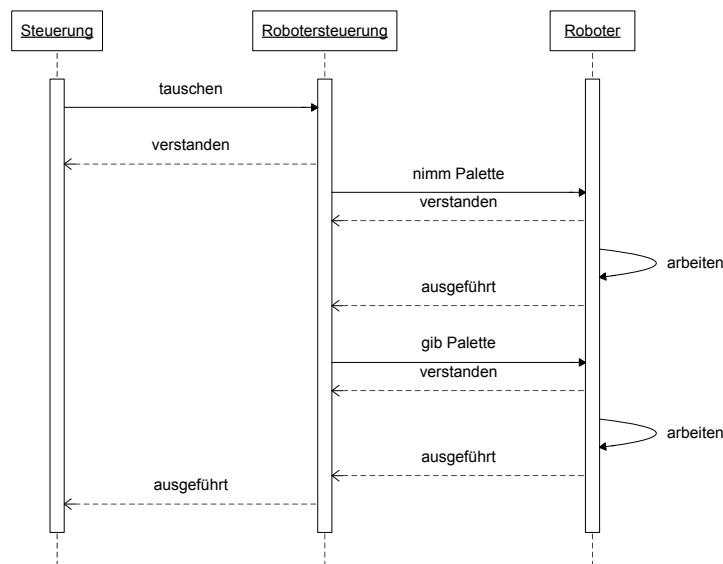


Abbildung 4.38: tauschen

4.2 Roboter

Dieser Befehl führt 2 interne Elementarbefehle hintereinander aus.

Zuerst sendet ST an Sr diesen Befehl. Nach der Antwort ob der Befehl verstanden wurde, sendet Sr an ro den Elementarbefehl eine LP vom Band auf einen freien Stellplatz (A oder B) zu legen, und die LP von dem anderen Platz auf T zu platzieren. Die Antworten bei diesem Befehl sind ebenfalls analog zu den anderen Befehlen.

4.2.8 Reset/Anfangszustand herstellen (Herunterfahren)

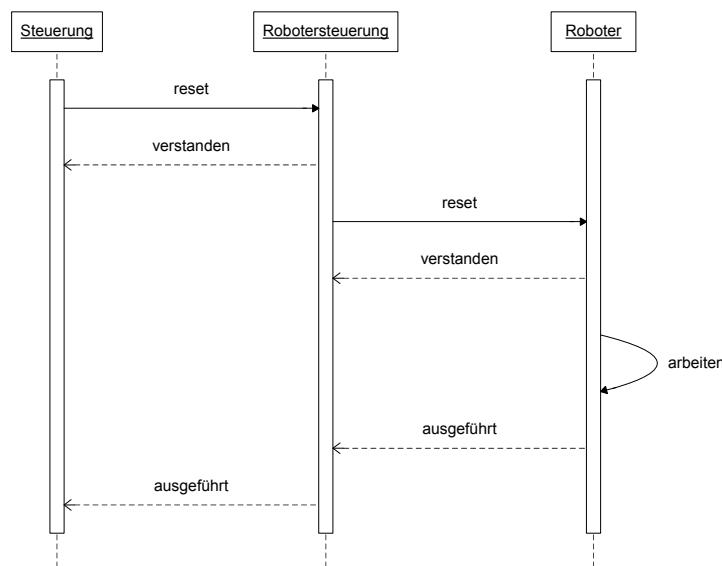


Abbildung 4.39: Reset/Anfangszustand (Herunterfahren)

Dieser Befehl setzt den Roboter von einem definierten Zustand in den Ausgangszustand zurück. Die Antwortstruktur ist gleich zu der aller vorherigen Befehle. Die Robotersteuerung wird heruntergefahren.

Siehe Beschreibung des Logfile (??) für mehr Erläuterungen.

4.3 Lagersystem

4.3 Lagersystem

Dieser Abschnitt beschreibt die Aufgaben des Lagersystems, und Interaktionen mit dem Steuerungssystem aus den Use-Case-Diagrammen, in ihrem chronologischen Ablauf in Form von Ablauf- und Sequenzdiagrammen. Die folgenden Beschreibungen gehen zunächst auf die Abläufe ohne Fehlerfälle ein. Mögliche Fehlerfälle bei den einzelnen Abläufen werden in einem eigenen Kapitel erläutert. Weiterhin werden Vereinbarungen und Aufgaben beschrieben, die im Rahmen der Anpassung an die neue Spezifikation an der bestehenden Software des Lagers notwendig sind.

4.3 Lagersystem

4.3.1 Abläufe Lagersteuerung und Lager

4.3.1.1 Hochfahren und Anmelden am System



Abbildung 4.40: Hochfahren des Lagersystems und Anmelden am System

Beim Hochfahren des Systems wird zuerst der letzte Lagerzustand eingelesen. Dieser beinhaltet den Smartie-Bestand und die Belegung der Paletten innerhalb des Lagers. Weiterhin versucht die Lagersteuerung eine Verbindung zum Lager über Port 30000 herzustellen. Danach meldet sich die Lagersteuerung bei der Steuerung über Port 40002 an. Als nächstes wird der letzte Bearbeitungstatus des letzten Befehls geprüft. Wenn er abgeschlossen war, ist das Lager bereit

4.3 Lagersystem

für einen neuen Auftrag. Sollte der letzte Befehl nicht ordnungsgemäß abgeschlossen worden sein, so versucht die Lagersteuerung in bestimmten Fällen unter Einbeziehung des Lageroperators den Befehl erneut durchzuführen. Sollte die Ausführung nun erfolgreich sein, so schickt die Lagersteuerung der Steuerung eine Quittung, dass die Ausführung erfolgreich war. Ansonsten schickt die Lagersteuerung eine Fehlermeldung (??).

4.3.1.2 Bestandsanfrage für eine Farbe

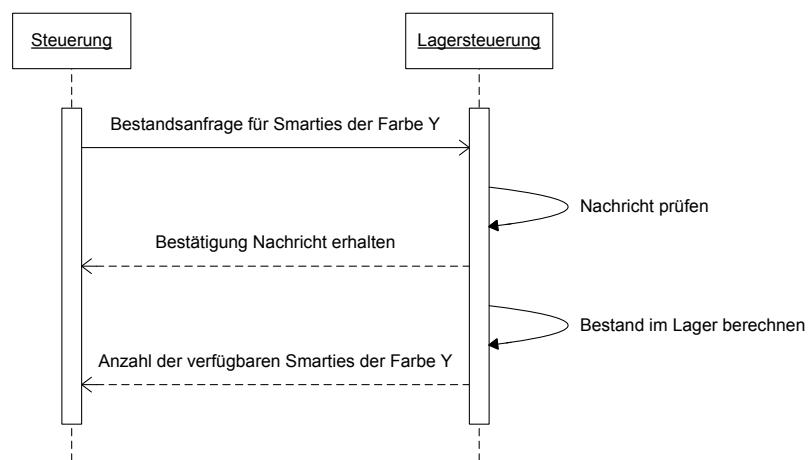


Abbildung 4.41: Bestandsanfrage für eine Farbe

Wenn die Lagersteuerung eine Anfrage für den Bestand einer Farbe erhält, prüft sie zunächst den Aufbau des Kommandos. Das Kommando enthält die Farbe, für die der Bestand ermittelt werden soll. Wenn das Kommando korrekt ist, wird der Bestand der entsprechenden Farbe über die in der Bestandsdatei vorhandenen Informationen ausgezählt. Der Bestand wird der Steuerung über die Quittung mitgeteilt.

4.3 Lagersystem

4.3.1.3 Anfrage auf Verfügbarkeit einer Smartiefarbe

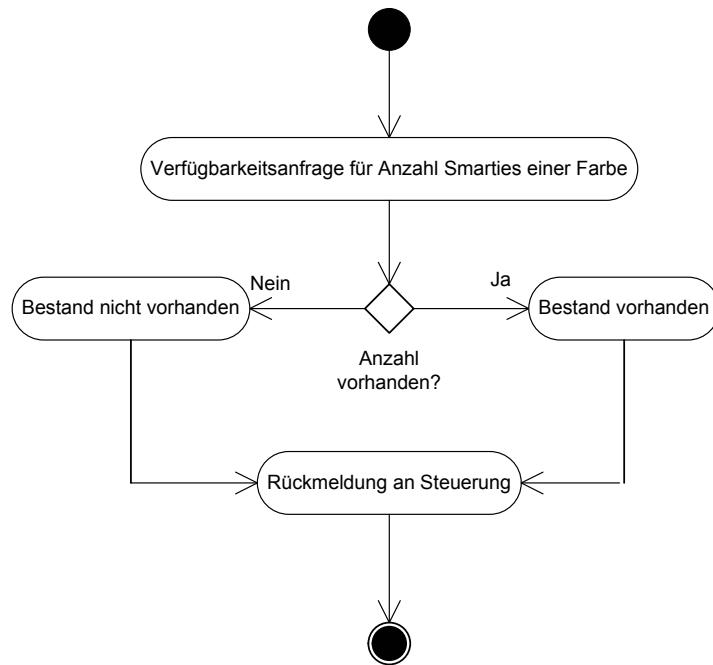


Abbildung 4.42: Anfrage auf Verfügbarkeit einer Smartiefarbe

Wenn die Lagersteuerung die Anfrage nach dem Bestand einer bestimmten Farbe bekommt, wird geprüft, ob die entsprechende Farbe in ausreichender Menge vorhanden ist. Die Anfrage enthält die Farbe und die benötigte Anzahl der Smarties. Danach schickt die Lagersteuerung an die Steuerung eine Rückmeldung, ob genügend Smarties der geforderten Farbe vorhanden sind.

4.3 Lagersystem

4.3.1.4 Anfrage Anzahl freie Stellplätze im Lager

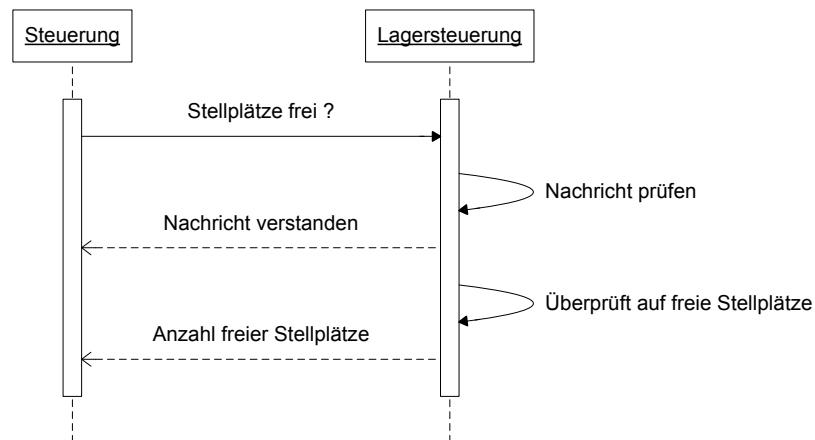


Abbildung 4.43: Anfrage freie Stellplätze im Lager

Wenn von der Steuerung die Anfrage nach der Anzahl der freien Stellplätze kommt, überprüft die Lagersteuerung wie viele Stellplätze noch frei sind. Diese Anzahl wird der Steuerung dann geschickt.

4.3 Lagersystem

4.3.1.5 Auftrag zur Einlagerung einer Palette

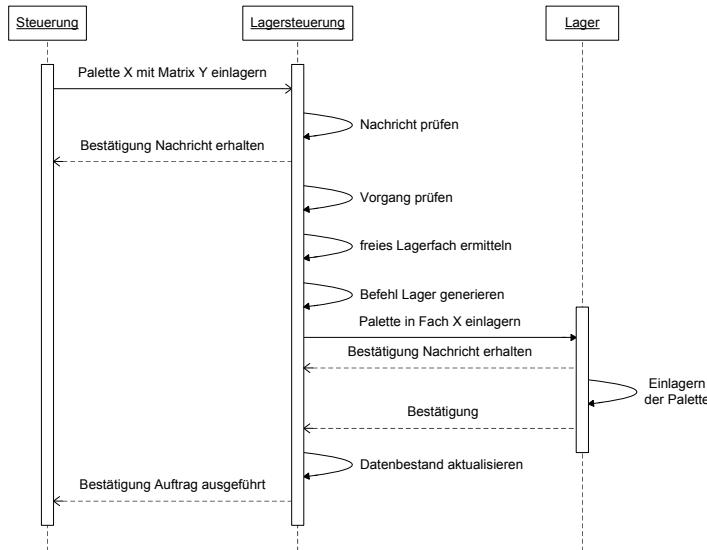


Abbildung 4.44: Auftrag zur Einlagerung einer Palette

Die Steuerung schickt einen Auftrag zur Einlagerung einer Palette an die Lagersteuerung. Der Auftrag enthält die Palettennummer und die spezifische Smartiebelegung der Palette. Nach Empfang des Auftrags prüft die Lagersteuerung über die Länge im Header der Nachricht, ob diese syntaktisch korrekt ist. Weiterhin wird geprüft, ob das Kommando bekannt ist. Nach erfolgreicher Prüfung quittiert die Lagersteuerung den Empfang und das Akzeptieren der Nachricht.

Das Lagersystem prüft nun, ob das Lagersystem aktuell in der Lage ist, den geforderten Vorgang auszuführen. Für die Einlagerung einer Palette muss das Lagersystem fehlerfrei sein, d.h. sich in keinem Fehlerzustand befinden, und es muss mindestens ein freier Stellplatz innerhalb des Lagers zur Verfügung stehen. Die Anzahl der freien Stellplätze wird beim Hochfahren aus der Bestandsdatei eingelesen.

Nach Prüfung des Auftrags auf Ausführbarkeit wird die Fachnummer des nächst freien Stellplatzes aus der Datenbasis ermittelt. Die Fachnummer wird anschließend für den Aufbau des Elementarbefehls an das Lager verwendet. Bevor das Lager den Auftrag abarbeitet, prüft das Lager die Nachricht und quittiert der Lagersteuerung den Erhalt und das Verständnis des Auftrags.

Das Lager bestätigt den Auftragsempfang und transportiert die Palette vom Schlitten des Transportsystems in das angegebene Fach und bestätigt die Ausführung an die Lagersteuerung. Erst jetzt wird der Datenbestand aktualisiert. D.h. die zwischengespeicherten Auftragsdaten, Palettennummer und Smartiebelegung, werden zu dem Fach ergänzt. Weiterhin wird der Zähler für die Anzahl der freien Stellplätze um einen Platz erniedrigt und in der Datenbasis gespeichert. Die erfolgreiche Durchführung des Auftrags an die Steuerung gemeldet.

4.3 Lagersystem

4.3.1.6 Auftrag zur Auslagerung einer Palette

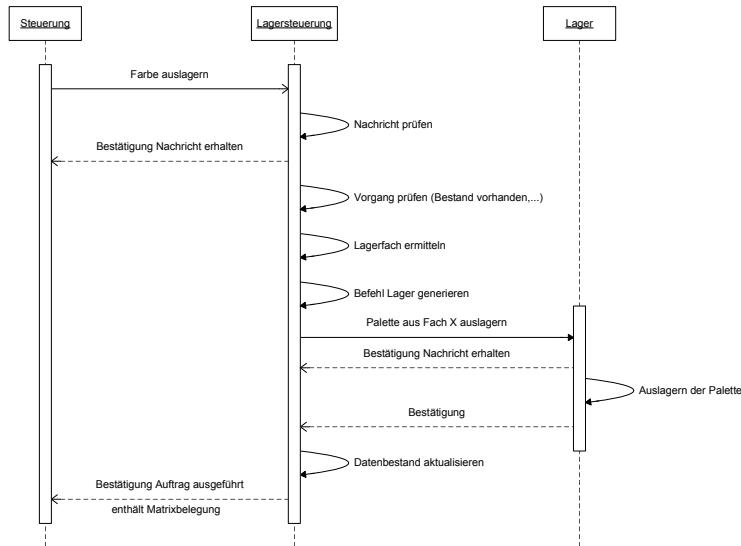


Abbildung 4.45: Auftrag zur Auslagerung einer Palette

Werden zur Bestückung der Produktpalette Smarties einer bestimmten Farbe benötigt, wird die Lagersteuerung von der zentralen Steuerung angewiesen, eine Lagerpalette mit den entsprechenden Smarties auszulagern. Ein Auslagerungsbefehl enthält immer die Farbe der benötigten Smarties, sowie deren Anzahl.

Um sicher zu gehen, dass ein gültiger Auslagerungsbefehl vorliegt, wird er zunächst auf seine Richtigkeit geprüft. Des Weiteren wird der Steuerungseinheit das Erhalten der Nachricht positiv quittiert.

4.3 Lagersystem

Als nächstes stellt die Lagersteuerung durch Auslesen der Bestandsdatei fest, ob sich genügend Smarties mit der angeforderten Farbe im Lager befinden. Wenn die Anzahl der sich im Lager befindlichen Smarties ausreicht, wird das Fach der auszulagernden Palette aufgrund der in Kapitel beschriebenen Regeln ausgewählt. Um den Gesamtauftrag korrekt abarbeiten zu können, spielt es keine Rolle, ob sich alle benötigten Smarties einer bestimmten Farbe auf einer einzigen Palette befinden, oder ob sie auf mehrere Paletten verteilt sind. Falls sie auf mehrere Paletten verteilt sind, wird dies von der Steuerung erkannt (Belegung der Matrix) und zieht einen weiteren Auslagerungsauftrag über die restlichen Smarties nach sich. Nachdem das entsprechende Lagerfach ermittelt wurde, kann nun für das Palettenlager der Elementarbefehl zum Auslagern generiert werden. Danach wird dieser an das Palettenlager geschickt. Das Erhalten, sowie die Ausführbarkeit des Elementarbefehls, wird jeweils mit einer entsprechenden Antwort quittiert. Nachdem die physikalische Auslagerung der Palette vom Palettenlager durchgeführt wurde, erhält die Lagersteuerung auch dafür die entsprechende Bestätigung. Anschließend aktualisiert die Lagersteuerung den Datenbestand. Dazu werden die entsprechenden Einträge der Bestandsdatei gemäß deren Spezifikation geändert.

Anschließend kann der Steuerung die erfolgreiche Ausführung des Auslagerungsbefehls gemeldet werden. Die Nachricht enthält außerdem die Palettennummer, sowie die Belegung der Matrix. Der Auslagerungsvorgang ist somit abgeschlossen.

4.3.1.7 Auftrag zum Herunterfahren des Systems

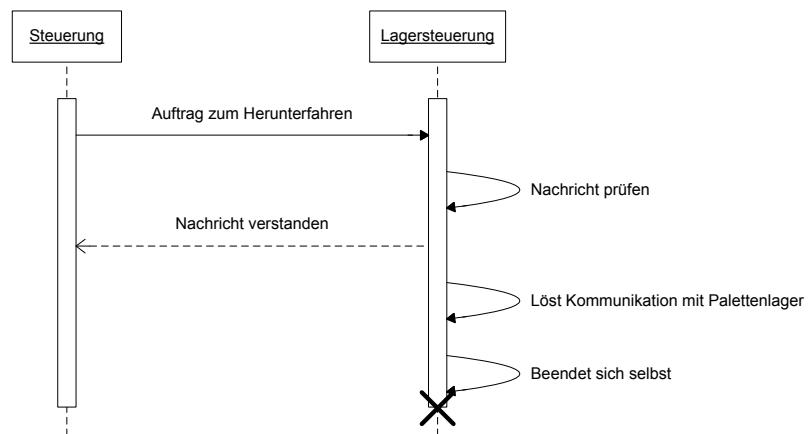


Abbildung 4.46: Auftrag zum Herunterfahren des Lagersystems

Nach Abarbeitung verschiedener Aufträge oder aufgrund eines Fehlers kann die Steuerung den Befehl zum Herunterfahren des Systems an die Lagersteuerung schicken. Die Lagersteuerung

4.3 Lagersystem

prüft die Nachricht auf deren korrekten Aufbau. Wurde die Nachricht verstanden, wird eine entsprechende Quittung an die Steuerung geschickt. Danach löst die Lagersteuerung die Kommunikation mit dem Palettenlager auf und beendet sich selbst. Die Bestandsdaten sowie der Zustand des Lagers müssen beim Herunterfahren nicht gesichert werden, da diese Daten nach jeder Änderung in der Datenbasis festgeschrieben werden. Außerdem wird hier natürlich keine Ausführungsbestätigung an die Steuerung geschickt, da das Lagersystem nach dem Beenden nicht mehr in der Lage dazu ist.

4.3.2 Anpassungen und Vereinbarungen an bestehende Lagerabläufe

4.3.2.1 Allgemeine Vereinbarungen

Es wurde beschlossen, zwei Module einmal für alle Systeme einheitlich zu konstruieren und zu implementieren.

- Modul "fdzNetwork"
- Modul "fdzMessageHandler"

Im weiteren Verlauf der Beschreibung wird nur der Einsatz dieser Module erläutert. Die genaue Funktionsbeschreibung ist der entsprechenden Dokumentation zu entnehmen.

4.3.2.2 Entfernung des früheren GUI-Codes aus dem Lager

Die übernommene Software enthält den Aufbau und die Verwaltung von zwei Port-Socket-Verbindungen. Erstere ist vergleichbar mit der Kommunikation zur Lagersteuerung, die zweite diente zur Anbindung einer Funktions- und Statusanzeige (im Code: GUI), die in der Spezifikation WS2005/2006 vorgesehen war. Da eine solche GUI nicht im Rahmen der aktuellen Spezifikation vorgesehen ist, soll die GUI zur besseren Lesbarkeit aus dem Code entfernt werden.

4.3.2.3 Beibehaltung Bearbeitungslogik durch zwei Tasks

In Hinblick auf spätere Erweiterungen der Lagersoftware durch weitere Tasks (z.B. eine GUI-Anbindung) wurde beschlossen die Bearbeitungslogik über zwei Tasks beizubehalten. Die Bearbeitung und Überwachung eines Auftrags der Lagerhardware soll nach wie vor im Task 1 stattfinden, die Überwachung des Netzwerks und die Ablauflogik zur Quittierung von Aufträgen in Task 2. Die Taskkoordination muss allerdings aufgrund der fest vorgegebenen Funktionsweise der einheitlichen Module zur Netzwerkkommunikation und Starten von Aufträgen über Callbacks des "fdzMessageParser" angepasst werden. Eine genaue Beschreibung der Taskinteraktion über diverse Steuerflags ist im Punkt "Taskinteraktion" beschrieben.

4.3 Lagersystem

4.3.2.4 Anpassung Fehlerbehandlung

Da aufgrund der Vereinheitlichung des Netzwerks- und Kommandoparsermoduls Netzwerkfehler und Fehler beim Parsen einer empfangenen Nachricht die jeweils eigenen Mechanismen genutzt werden sollen, müssen diese in die angepasste Auftragslogik integriert werden. Die Fehlerbehandlungsmimik für auftretende Fehler innerhalb der Hardware über eine Fehlerqueue kann weitestgehend beibehalten werden, der Bearbeitungszeitpunkt muss aber im Rahmen der Veränderung der Taskbearbeitung angepasst werden.

4.3.2.5 Anpassung des Recovery an das neue Protokoll

Die Mimik zum Recovery eines Auftrags ist bereits in der bestehenden Software umgesetzt. Sie muss an das neue Protokoll angepasst werden und neu die veränderte Bearbeitungslogik durch zwei Threads integriert werden. Eine genau Beschreibung dieser Abläufe befinden sich im Abschnitt (??).

4.3.2.6 Integration der Lichtschranke zur Lagerfachprüfung

Im Laufe des letzten Jahres wurde die Lagerhardware um eine Lichtschranke zur Prüfung des angefahrenen Lagerfaches erweitert. Die Lichtschranke wurde offensichtlich nachträglich in den Code aufgenommen, da sie ausschließlich den Auftrag stoppt und eine Fehlermeldung am Display ausgibt. Eine entsprechende Fehlermeldung an die alte Lagersteuerung wird nicht gesendet. In dem überarbeiteten System soll die Lichtschranke zu Beginn eines Auftrags prüfen, ob bei einer Auslagerung das entsprechende Fach auch gefüllt ist, und ob bei einer Einlagerung das angefahrene Fach auch leer ist. Ist dem jeweils nicht der Fall, soll neben einer Fehlerausgabe an Konsole und LCD auch eine Fehlermeldung (Fehler beim Ein-/Auslagern, F001/F002) an die Lagersteuerung gesendet werden.

4.3.2.7 Überarbeitung des Codes

In weiten Teilen des Codes befinden sich sehr viele kommentarlos auskommentierte Codefragmente. Diese tragen nicht zur Lesbarkeit bei und können entfernt werden. Außerdem wurde bei intensiver Analyse des Codes festgestellt, dass viele Variablen und auch Funktionen ohne Funktion sind. Diese sollen im Rahmen der Überarbeitung entfernt werden.

4.4 Transportsystem

4.4.1 Bestimmten Schlitten an bestimmter Position anfordern

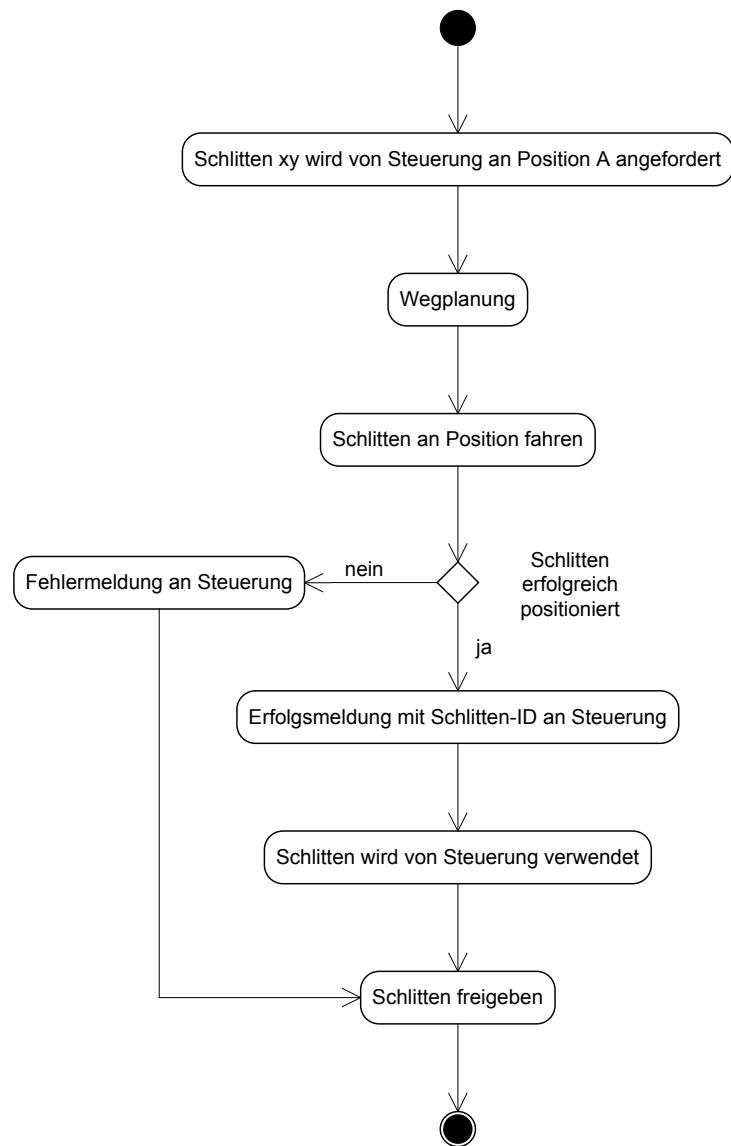


Abbildung 4.47: Bestimmten Schlitten anfordern

Wenn Steuerung das Kommando schickt, einen bestimmten Schlitten an eine bestimmte Position zu bringen, muss die Bandsteuerung zunächst eine Art Wegplanung (siehe Abschnitt ??) durchführen. Prinzipiell wird dabei die bestmögliche Route vom derzeitigen Standort des Schlittens zur gewünschten Position bestimmt. Anschließend wird dieser Schlitten an diese Position

4.4 Transportsystem

gebracht. Dafür ist das Transportband zuständig. Bandsteuerung benutzt zu diesem Zweck das Band, die Weichen und die Sperren des Transportbandes.

Man beachte jedoch, dass es sein kann, dass der Schlitten aus irgendwelchen Gründen nicht ankommt (siehe den Abschnitt ?? "Fehlermeldungen"). Dies kann anhand der Bandgeschwindigkeit und der Sensoren erkannt werden. Ist dies der Fall, wird eine Fehler-Nachricht an Steuerung gesendet. Ist der Schlitten hingegen an seiner Bestimmungsposition angekommen, meldet Bandsteuerung einen Erfolg an Steuerung, zusammen mit der Schlitten-Nummer (Schlitten-ID).

Der Schlitten wird danach weiter von Steuerung verwendet, dabei handelt es sich im Prinzip wieder um die gleichen Abläufe, die sequentiell (mit diesem Schlitten) ablaufen. Beispielsweise könnte Steuerung den Schlitten als nächstes an einer anderen Position anfordern. Sobald der Schlitten jedoch nicht mehr verwendet wird (d.h. in der Regel, dass er nichts mehr transportiert), wird er von Steuerung freigegeben. Er kann dann von Bandsteuerung für die Anforderung eines leeren Schlittens (siehe Aktivitätsdiagramm ??) verwendet werden.

4.4.2 Einen leeren Schlitten anfordern

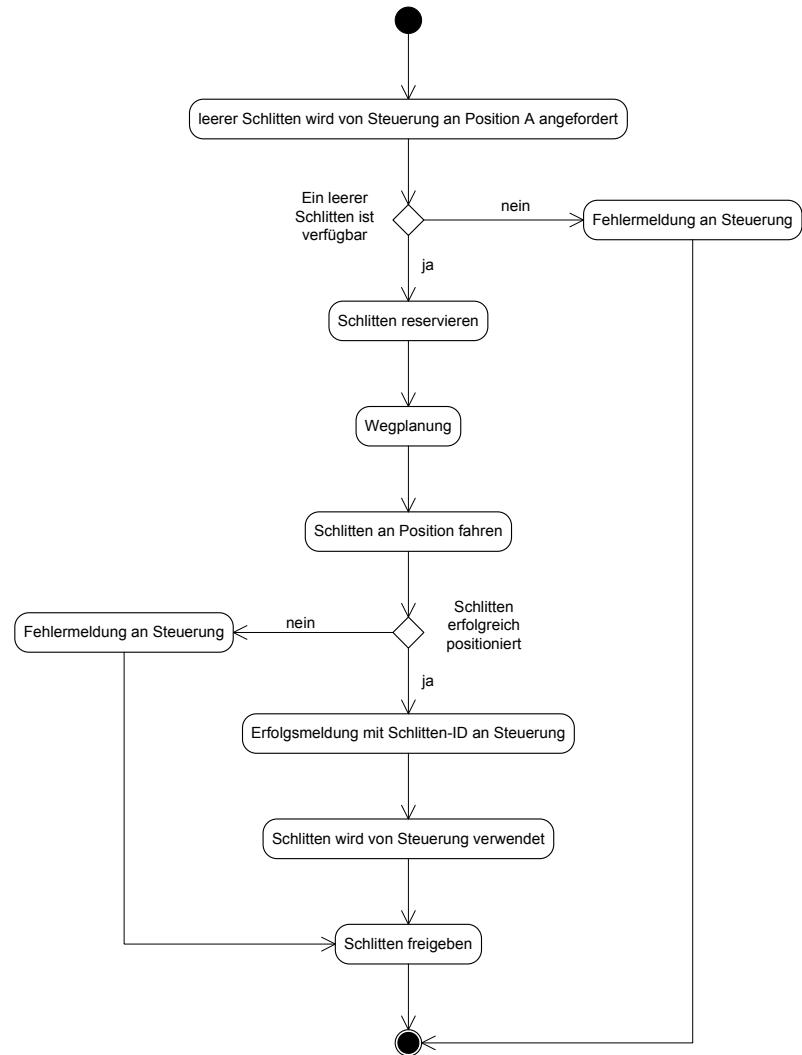


Abbildung 4.48: Leeren Schlitten anfordern

Dieses Aktivitätsdiagramm ähnelt stark dem Aktivitätsdiagramm ???. Der Unterschied hierbei ist, dass es sich um einen separaten Befehl von Steuerung handelt, der die Schlitten-Nummer nicht angibt. Außerdem handelt es sich immer um einen nicht beladenen Schlitten, der dabei verwendet wird.

Bandsteuerung markiert den Schlitten sofort als belegt, so dass er für darauf folgende Anforderungen eines leeren Schlittens nicht mehr zur Verfügung steht. Danach läuft der gleiche Prozess

4.4 Transportsystem

ab, wie bei der Anforderung eines bestimmten Schlittens: Wegplanung - d.h. Finden einer Route - und das Positionieren des Schlittens. Wieder kann es sein, dass der Schlitten entweder an seinem Ziel ankommt oder nicht. Kommt er nicht an, gibt es eine Fehlermeldung an Steuerung, andernfalls gibt es eine Erfolgsmeldung. Zusammen mit dieser Erfolgsmeldung wird die Schlitten-ID des leeren Schlittens erstmals an Steuerung bekannt gegeben, so dass dieser im Folgenden - wie im Aktivitätsdiagramm ?? dargestellt - verwendet werden kann.

Nachdem Steuerung den Schlitten nicht mehr benötigt (d.h. in der Regel, dass er nicht mehr beladen ist), muss Steuerung eine Nachricht an Bandsteuerung schicken, die bewirkt, dass der Schlitten freigegeben wird und wieder angefordert werden kann.

4.4 Transportsystem

4.4.3 Schlitten-Handling

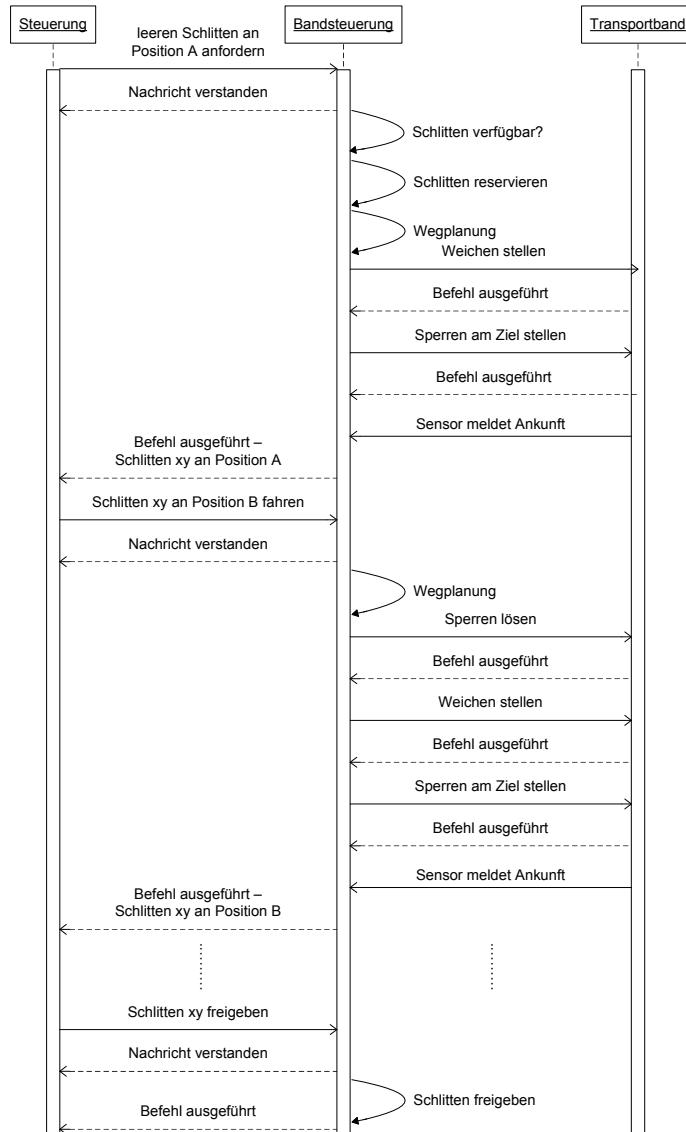


Abbildung 4.49: Schlitten-Handling

In diesem Sequenzdiagramm werden gleich beide Aktivitätsdiagramme von oben verwertet. Zu aller erst wird ein leerer Schlitten von Steuerung angefordert (Aktivitätsdiagramm ??), dann geht es weiter wie im Aktivitätsdiagramm ?. Wurde der leere Schlitten an die gewünschte Position gebracht, kennt Steuerung nun auch die ID dieses Schlittens. In der Regel wird Steuerung daraufhin Nachrichten zur Positionierung dieses bestimmten Schlittens senden, wie hier dargestellt ("an Position B fahren").

Irgendwann ist Steuerung mit diesem Schlitten fertig und gibt ihn wieder frei.

4.4.4 Wegplanung

Im folgenden Diagramm wird das Prinzip einer einfachen Wegplanung dargestellt.

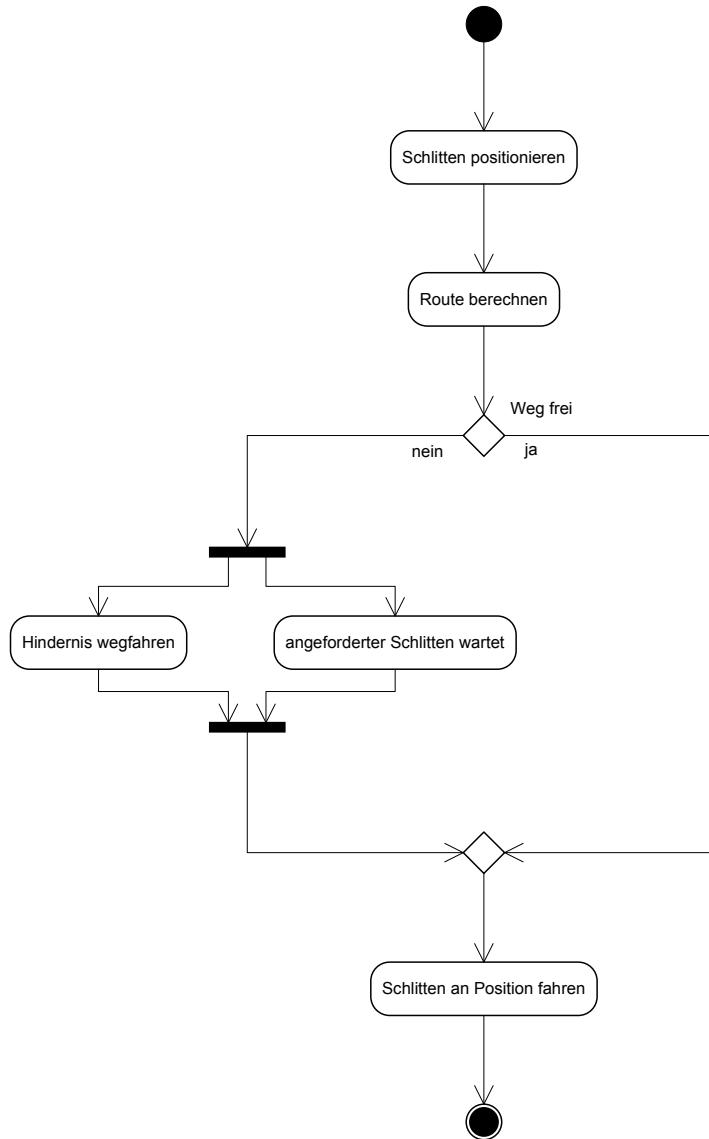


Abbildung 4.50: Wegplanung

Wenn ein Schlitten positioniert werden soll, muss zunächst eine Route berechnet werden. D.h. es wird anhand der Position anderer Schlitten und der Zielposition eine Route gesucht, die frei ist. Dabei ist anhand des Aufbaus des Bandes, der Lage der Stationen und der Bewegungsrichtung des Bandes klar, dass es hier nicht viele Möglichkeiten gibt (im Prinzip gibt es sowieso immer nur einen Weg).

4.4 Transportsystem

Ist zwischen Start- und Zielposition auf der Route ein Hindernis (also ein anderer Schlitten), muss dieses erst weggefahren werden. Dabei kann es sein, dass das Hindernis seinerseits bereits benötigt wird, also reserviert ist. In diesem Fall kann das Hindernis natürlich erst weggefahren werden, wenn es nicht mehr benötigt bzw. ohnehin bald weggefahren wird. Wenn das Hindernis nicht reserviert ist, d.h. zur Zeit nicht benutzt wird, kann es ohne Probleme weggefahren werden.

Natürlich muss die gleiche Wegplanung auch wieder für dieses Hindernis durchgeführt werden, und für alle weiteren potentiellen Hindernisse.

Möglicherweise kommt es bei dieser Vorgehensweise irgendwann zu einer nicht enden wollenden Rekursion, wenn sich nur genügend Schlitten auf dem Band befinden. Eine weitere Folge zu vieler Schlitten könnten Deadlocks, also nicht entwirrbare Staus, sein. Um diese Probleme zu vermeiden, könnte man noch die Auslauf-Schleifen (Band 2 und 3) verwenden.

Das Grundprinzip dieses Ansatzes ist es, dass der Schlitten, der positioniert werden soll, so lange warten muss, bis eventuelle Hindernisse aus dem Weg geräumt sind. Erst dann kann er an die jeweilige Position gefahren werden.

4.5 Eingabe/Ausgabe-Station

4.5.1 Aus und Einschleusen einer Palette

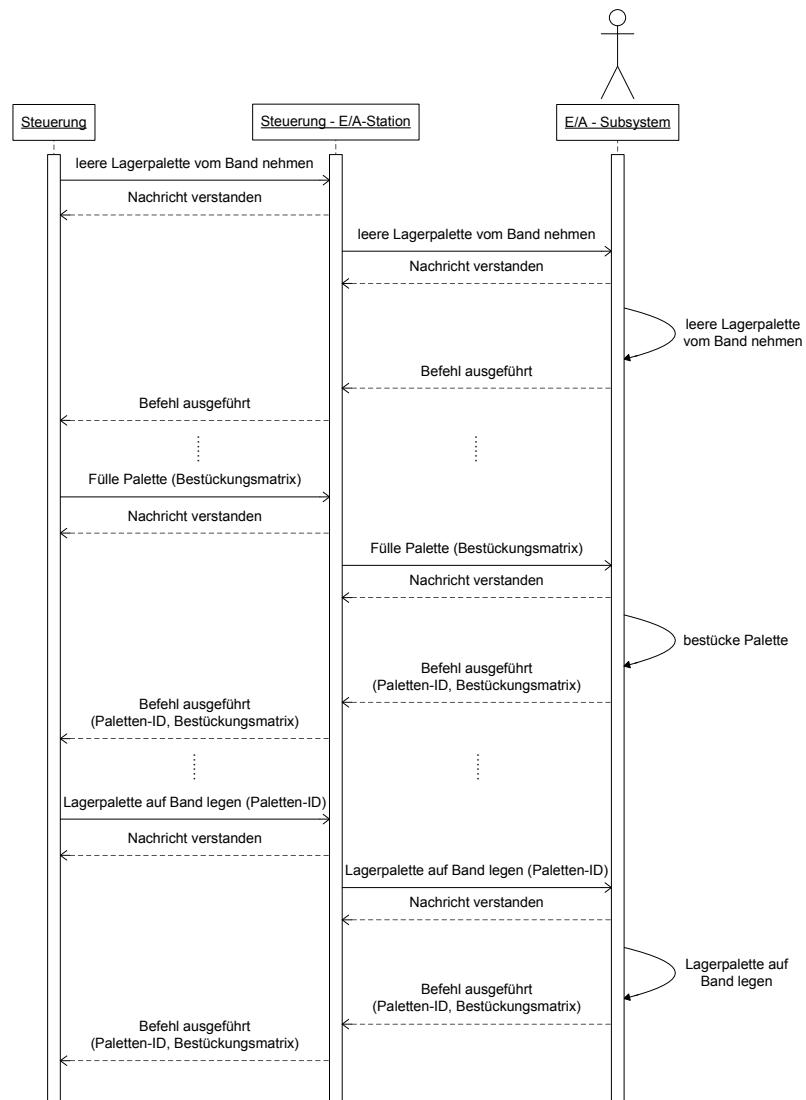


Abbildung 4.51: Palette ein- und ausschleusen

In diesem Diagramm wird das Aus- und Einschleusen einer Palette dargestellt. Der Vorgang des Ausschleusens lässt sich sowohl auf eine leere Lagerpalette als auch auf eine bestückte Produktpalette anwenden. Das Einschleusen ist hier speziell für eine Lagerpalette dargestellt.

Zunächst zum Ausschleusen: Steuerung schickt eine Nachricht an Bandsteuerung, dass eine Palette ausgeschleust werden soll. Nach dem Versenden des ersten Acknowledge an Steue-

4.5 Eingabe/Ausgabe-Station

rung (Nachricht verstanden) teilt Steuerung E/A-Station dem untergeordneten System (zur Zeit ein Mensch) mit, dass die Palette vom Band genommen werden soll. Wurde dies vom E/A-Subsystem erledigt, wird ein entsprechendes Acknowledge an Steuerung E/A-Station geschickt, worauf natürlich ein Acknowledge an Steuerung folgt (Befehl ausgeführt).

Das Einschleusen findet zu einem anderen Zeitpunkt unabhängig vom Ausschleusen statt, wurde aber aus praktischen Gründen gleich mit in dasselbe Sequenzdiagramm eingezeichnet. Zunächst muss Steuerung einen Füllen-Befehl an die Steuerung E/A-Station senden, in dem sie die gewünschte Bestückung mitteilt. Steuerung-E/A-Station leitet diese Anforderung an das E/A-Subsystem weiter. Wurde die Palette erfolgreich bestückt, liefert das Subsystem zusammen mit dem Acknowledge die ID der bestückten Palette und die tatsächliche Bestückung zurück. Dieses Acknowledge wird an Steuerung weitergereicht. Daraufhin können beliebige andere Kommandos gegeben und ausgeführt werden. Zu einem späteren Zeitpunkt schickt Steuerung den Befehl, die vorher befüllte Lagerpalette einzuschleusen, wobei sie deren Paletten-ID angeben muss. Steuerung E/A-Station schickt nach dem Acknowledge an Steuerung (Befehl verstanden) einen entsprechenden Befehl an das untergeordnete System, die Palette mit ID xy aufs Band zu legen, was auch ausgeführt wird. Zuletzt werden entsprechende Acknowledges (Befehl ausgeführt) an das jeweils übergeordnete Systeme geschickt. Diese enthalten wiederum die Paletten-ID und die tatsächliche Bestückung als Parameter.

5 Nachrichtenpezifikation

5.1 Das Protokoll

Das Gesamtsystem *Fabrik der Zukunft* ist nach dem Client–Server–Prinzip aufgebaut. Informationen werden dabei über eine TCP/IP basierte Port–Socket–Kommunikation ausgetauscht.

In dem folgenden Abschnitt wird das für die Kommunikation verwendete Nachrichtenformat spezifiziert. Daran anschließend werden in weiteren Abschnitten alle verwendeten Nachrichten aufgeführt, ihre Verwendung erläutert und es wird eine Verknüpfung zu den dazugehörigen Use–Case und Interaktionsdiagrammen aufgebaut.

5.1.1 Protokollaufbau

Jede Nachricht ist aus zwei Teilen, dem Protokollheader und den optionalen Nutzdaten, aufgebaut. Sowohl die im Protokollheader als auch in den Nutzdaten abgelegten Informationen werden durch lesbare ASCII–Texte kodiert. Dadurch kann man mit Einsatz entsprechender Low–Level Diagnosetools (zum Beispiel `tcpdump`) den gesamten Nachrichtenaustausch zwischen den Systemen mitlesen.

5.1.1.1 Protokollheader

Der Protokollheader besteht aus 25 Bytes, er ist wie folgt aufgebaut:

Position	Länge	Beschreibung
0	2	Absender der Nachricht
2	2	Empfänger der Nachricht
4	1	Nachrichtentyp
5	3	Befehlsnummer
8	13	Eindeutige Message–Id
21	4	Länge der Nutzdaten

Tabelle 5.1: Aufbau des Protokollheaders

Im *Absender*–Feld der Nachricht ist der Absender der Nachricht durch eine Kennung festgelegt, analog dazu wird im *Empfänger*–Feld der Empfänger der Nachricht festgelegt. Die Tabelle ?? beschreibt die bisher verwendeten Kennungen und listet die dazugehörigen Systeme auf.

5.1 Das Protokoll

Kennung	Beschreibung
JS	Java–Interface zum SAP–System
ST	Steuerungsprogramm der Fertigungszelle
Sr	Steuerungsprogramm des Roboters
St	Steuerungsprogramm des Transportsystems
SI	Steuerungsprogramm des Lagers
Se	Steuerungsprogramm des E/A–Systems
ro	Robotersystem
tr	Transportsystem
la	Lagersystem
ea	E/A–System
GU	Graphische Bedienoberfläche zur Eingabe einer Bestellung

Tabelle 5.2: Kodierungen für den Sender und den Empfänger

Im *Nachrichtentyp* wird hinterlegt um welche Art einer Nachricht es sich handelt. In der Tabelle ?? sind die definierten Kennungen K: „Kommando“

- , A: „Antwort“
- , F: „Fehler“
- , aufgeführt.

Kennung	Beschreibung
K	Kommando
A	Antwort
F	Fehler

Tabelle 5.3: Kodierung des Nachrichtentyps

Die einzelnen Befehle sind durch *Befehlsnummern* gekennzeichnet, diese Kennzeichnung erfolgt eindeutig durch Nummern zwischen 0 und 999. Die Befehlsnummer wird als ASCII–codierte Darstellung übertragen, dabei wird die Zahl rechtsbündig im Feld abgelegt, Leerstellen müssen mit einer 0 aufgefüllt werden. Die Nachricht vom Typ 42 hat somit 042 als Darstellung. Eine Beschreibung aller Nachrichten ist im Abschnitt ?? enthalten

Jede Nachricht die zwischen den einzelnen Systemen ausgetauscht wird erhält eine eindeutige Message–Id. Diese Message–Id setzt sich aus zwei Teilen zusammen:

1. Zeit in Sekunden seit dem 1.1.1970. Dieser Wert wird in einem 10 stelligen Feld abgelegt, dazu wird die ASCII–codierte Darstellung der Zahl verwendet. Unix–Systeme stellen diesen Wert durch den Aufruf der Funktion `time()` in C Programmen zur Verfügung.
2. In dem zweiten Teil werden die in der aktuelle Sekunde verschickten Nachrichten durchnummieriert. Die erste Nachricht wird dabei durch die Zeichenkette 00 gekennzeichnet.

Die beiden Teile werden durch einen Doppelpunkt (:) voneinander getrennt.

5.1 Das Protokoll

Im letzten Feld des Protokollheaders wird die Länge des optionalen Teils der Nutzdaten abgelegt. Die Länge wird dabei in einer ASCII—codierten Darstellung übertragen. Der Wert wird rechtsbündig abgelegt, führende Leerstellen werden durch eine 0 aufgefüllt. Der theoretische Maximalwert für dieses Feld ist 9999. Dies entspricht jedoch nicht der maximalen Nachrichtenlänge. Diese wird auf 1024 Bytes festgelegt, um Speicher zu sparen. Werden keine Nutzdaten übertragen, enthält das Feld den Eintrag 0000.

5.1.1.2 Nutzdaten

Der zweite Teil der Nachricht ist für die Nutzdaten reserviert. Die Nutzdaten werden dabei durch lesbare ASCII-Zeichen repräsentiert, nur in Ausnahmefällen ist die Verwendung nicht lesbarer Zeichen erlaubt.

5.1.2 Nachrichtenaustausch

Um einen maximalen Sicherheit beim Nachrichtenaustausch zu gewährleisten, wird jede Nachricht die ein Kommando darstellt durch *zwei* Antworten bestätigt:

5.1.2.1 A001: Acknowledge 1

Wenn das Kommando empfangen wurde und auch interpretiert werden kann, wird eine Antwort mit der Befehlsnummer 001 an den Absender des Kommandos zurückgeschickt. Antworten sind dabei durch ein A als Nachrichtentyp gekennzeichnet.

Konnte das Kommando nicht korrekt interpretiert werden, wird eine Fehlernachricht mit der Befehlsnummer 000 zurück geschickt. Fehler sind durch ein F im Nachrichtentyp gekennzeichnet,

5.1.2.2 A002: Acknowledge 2

Sobald ein Kommando korrekt abgearbeitet wurde, wird dies durch das Versenden einer Antwort mit der Befehlsnummer 002 gekennzeichnet. Wenn zusätzlich Nutzdaten übertragen werden müssen, sind die entsprechenden Antworten bei der nachfolgenden Beschreibung der Nachrichten aufgeführt. Eine Antwort 002 ohne Nutzdaten stellt eine generelle Bestätigung der Abarbeitung des jeweiligen Befehls dar.

Konnte das Kommando nicht korrekt ausgeführt werden, wird eine entsprechende Fehlernachricht zurück geschickt. Auch hier können optional Nutzdaten zur näheren Beschreibung der Fehlersituation an die Nachricht angehängt werden. Diese Fehlerfälle werden in der folgenden Beschreibung der Nachrichten erläutert.

Bei Antworten oder Fehlermeldungen wird für die Nachricht keine eigene Message-ID generiert, vielmehr wird die Message-ID des zugehörigen Kommandos verwendet. Somit lassen sich die Antworten und Fehlermeldungen eindeutig den entsprechenden Nachrichten zuordnen, ein paralleler Austausch von Nachrichten ist somit möglich.

5.1 Das Protokoll

5.1.3 Auflistung aller Nachrichten

In den folgenden Unterabschnitten werden alle Nachrichten, die im System Fabrik der Zukunft verwendet werden, aufgelistet und beschrieben.

Bei der Beschreibung der Nachrichten werden folgenden Abkürzungen verwendet:

- <Message--Id> steht dabei für die eindeutige Message-Id (siehe Abschnitt ?? auf Seite ??).
- {rgybw-}*63 beschreibt die Belegung einer Produktpalette. Jedes Zeichen beschreibt dabei die Belegung einer Position in der Palette. Während r (red), g (green), y (yellow), b (blue) und w (brown) die Farbe des jeweiligen Smarties bezeichnen, steht – für ein leeres Feld in der Produktpalette.
- {rgybw-}*91 beschreibt die Belegung einer Lagerpalette. Siehe vorhergehenden Punkt.
- {rgybw} beschreibt die Paletten-ID, welche die Farbe der Smarties auf der Lagerpalette angibt.
- {rgybw-}*63 beschreibt im Fehlerfall F006 bei einer Bestückung, welche Positionen der Produktpalette bereits belegt sind. Positionen die bestückt werden könnten, werden mit – gekennzeichnet. Positionen, welche belegt sind, aber nochmals bestückt werden sollten, werden mit der Farbe der belegten PP-Position markiert.
- {rgybwX-}*63 beschreibt als Antwort einer abgeschlossenen Bestückung die Matrix der abgearbeiteten Produktpalette. Die Matrix umfasst dabei immer nur die aktuell angeforderte Bestückung. Vorhergehende Bestückungen werden nicht berücksichtigt, d.h. die Matrix gibt nicht die vollständige PP-Belegung wieder. Erfolgreich bestückte Positionen werden mit der jeweiligen Farbe markiert. Sind auf der LP nicht genügend Smarties vorhanden, so werden die fehlenden, nicht bestückten Positionen mit X markiert.

5.2 Befehle des Java–SAP Interface an das Steuersystem

Das Java–SAP Interface sendet dem Steuersystem folgende Befehle:

- Der Start eines Bestückungsvorganges wird durch die Nachricht

JSSTK000<Message-ID>0063{rgybw-}*63

ausgelöst. Nach dem aus 23 Byte bestehenden Protokollheader wird die gewünschte Belegung der Produktpalette angegeben.

- Durch den Befehl

JSSTK001<Message-ID>0000

wird der aktuelle Lagerbestand der Smarties abgefragt.

- Der Abbruch eines in Bearbeitung befindlichen Auftrages wird durch das Kommando

JSSTK002<Message-ID>0000

ausgelöst.

5.3 Steuerung

5.3.1 Befehl- und Antwortliste

5.3.1.1 Kommunikation zwischen der GUI und dem Java–SAP Interface

Zwischen der GUI und dem Java–SAP Interface können folgenden Nachrichten ausgetauscht werden:

- Ein neuer Bestellauftrag wird durch den Befehl

GUJSK000<Message-ID>072{rgybw-}*63<User-Id>

von den graphischen Benutzerterminals in das System eingelesen. Neben dem Bestückungsmuster wird mit der Nachricht eine dem Besteller eindeutig zuordnbare, neunstellige Besteller–Kennung mitgeschickt.¹

Sobald der Auftrag korrekt eingelesen wurde, wird von dem Java–SAP Interface die Antwort

JSGUA001<Message-ID>0009xxxxxxxx

geschickt. In den neun Byte Nutzdaten wird die eindeutige Kennung des Auftrags innerhalb der Fabrik der Zukunft an den Client zurückgeschickt.

- Der Bearbeitungsstand eines bereits eingelesenen Auftrags kann durch das Kommando

GUJSK001<Message-ID>0009xxxxxxxx

¹In der aktuellen Version der FdZ wird nur ein Kunde unterstützt, seine Kundennummer ist 0000 00042

5.3 Steuerung

abgefragt werden.² In den neun Byte langen Nutzdaten wird die vom Java–SAP beim Einlasten eines Auftrags zurückgegebene Kennung angegeben. Das Ergebnis der Anfrage wird in der Antwort

JSGUA001<Message-ID>0006ttttt

an das GUI Programm zurückgeliefert. In den 6 Bytes der Nutzdaten wird dabei die voraussichtlich noch vergehende Zeit bis zum Ende der Bearbeitung der Bestellung in Sekunden zurückgeliefert.

5.3.1.2 Antworten vom Steuersystem an das Java–SAP Interface

Nach Erhalt der Befehle generiert die Steuerung folgende Antworten an das JSAP:

- Wurde der empfangene Befehl von der Steuerung verstanden, generiert diese folgende Antwort:

STJSA001<Message-ID>0000

- Eine Bestätigung, dass der Auftrag produziert werden kann, wird mit der Antwort

STJSA002<Message-ID>0000

an JSAP versandt.

- Sobald der Auftrag erfolgreich abgeschlossen wurde, wird eine Antwort in der Form

STJSA003<Message-ID>0018rxxgxxxyxxbxxwxxttt

verschickt. In den Nutzdaten ist dabei kodiert, wieviele rote, grüne, gelbe, blaue und braune Smarties tatsächlich verbraucht wurden. Durch die dreistellig codierte Zahl ttt wird die für die Abarbeitung des Bestückungsauftrag benötigte Zeit in Sekunden an das SAP–System gemeldet. Sollte die Bearbeitung länger als 999 Sekunden gedauert haben, wird der Wert 000 zurückgegeben.

- Wurde ein Auftrag auf Befehl von JSAP abgebrochen, erhält JSAP die Nachricht

STJSA004<Message-ID>0000

- Der Lagerstatus wird mit der Antwort

STJSA005<Message-ID>0020rxxxgxxxxyxxxxbxxxwxxxx

an JSAP zurück gegeben. In den Nutzdaten ist dabei kodiert, wieviele rote, grüne, gelbe, blaue und braune Smarties im Lager der Fertigungszelle vorhanden sind.

²Achtung: bei der im SS2006 implementierten Kommunikationsbibliothek entsprechend Abschnitt ?? und ?? wird diese Funktion nicht unterstützt, da die Bibliothek davon ausgeht, dass der Nachrichtenfluss streng dem Prinzip „ein Kommando bzw. eine Antwort“ folgt.

Um diese Funktionalität zu realisieren, muss entweder die Bibliothek erweitert oder eine zweite Netzwerkverbindung für die Statusabfragen geöffnet werden.

5.3 Steuerung

5.3.1.3 Fehlermeldungen vom Steuersystem an das Java–SAP Interface

Sobald Fehler bei der Bearbeitung der Kommandos auftreten, werden folgende Fehlerarten generiert:

- Wurde ein Befehl nicht verstanden, wird dies durch die Meldung

STJSF000<Message-ID>0000

dem JSAP signalisiert.

- Tritt ein Fehler bei der Produktion auf, wird dieser durch die Meldung

STJSF001<Message-ID>0015rxxgxxxyxxbxxwxx

dem JSAP mitgeteilt. Als Parameter wird die bisher verbrauchte Anzahl an Smarties, geordnet nach Farben, mitgegeben.

- Ist ein Auftrag auf Grund fehlender Smarties nicht produzierbar, erhält JSAP die Meldung

STJSF002<Message-ID>0000

- Werden Anfragen von JSAP während der Produktion gestellt, (z. B. "produziere Auftrag" oder "Lagerstatus abfragen"), werden diese mit der Meldung

STJSF003<Message-ID>0000

abgewiesen.

- Ist kein Auftrag vorhanden, erhält JSAP die Meldung

STJSF004<Message-ID>0000

Diese Meldung wird generiert, wenn die Steuerung den Befehl "Auftrag abbrechen" von JSAP empfängt.

- Ist der in Produktion befindliche Auftrag nicht mit dem identisch, der abgebrochen werden soll, erhält JSAP folgende Fehlermeldung

STJSF005<Message-ID>0000

5.4 Robotersystem

Folgend sind alle derzeit möglichen Befehle/Elementarbefehle und Antworten an die und innerhalb der beiden Robotersysteme aufgelistet. Für eine detaillierte Erklärung der Abläufe der einzelnen Vorgänge möchten wir auf die Kapitel ?? und ?? verweisen.

5.4.1 Befehl- und Antwortliste

Zuerst werden jeweils die Befehle bzw. Elementarbefehle aufgelistet, die an die Robotersteuerung und an das Robotersystem gesendet werden können. Anschliessend folgen deren Antworten an den jeweiligen Absender.

5.4.1.1 Befehle der Steuerung (ST) an die Robotersteuerung (Sr)

5.4.1.1.1 STSrK000: Nehme Lagerpalette vom Transportband:

Anweisung, eine Lagerpalette vom Transportband auf eine freie Position des Arbeitsbereiches des Roboters zu legen. Als Parameter an Sr wird die LP-Information übergeben, welche von Sr gespeichert wird.

STSrK000<Message-ID>0091{rgybw-}*91

Es werden keine Informationen in der Antwort an ST zurückgegeben.

5.4.1.1.2 STSrK001: Nehme Produktpalette vom Transportband:

Anweisung, die Produktpalette vom Transportband auf die Bestückungsposition des Arbeitsbereiches des Roboters zu legen. Sr benötigt hierfür keine Parameter.

STSrK001<Message-ID>0000

Es werden keine Informationen in der Antwort an ST zurückgegeben.

5.4.1.1.3 STSrK002: Bestücke Produktpalette:

Anweisung, die Produktpalette mit der übergebenen Produktmatrix zu bestücken. Sr teilt die übergebene Matrix in einzelne Elementarbefehle für ro auf und sendet diese an ro.

STSrK002<Message-ID>0063{rgybw-}*63

Nach dem Bestücken wird an ST die abgearbeitete PP-Matrix zurückgegeben.

5.4 Robotersystem

5.4.1.1.4 STSrK003: Stelle Lagerpalette auf das Transportband:

Anweisung, die Lagerpalette zurück auf das Transportband zu stellen. Es werden keine Parameter an Sr übergeben.

STSrK003<Message-ID>0000

Sr teilt ST die LP-Information in der Antwort mit. Da nie, ausser kurzzeitig beim Tauschen von Lagerpaletten, zwei Lagerpaletten auf dem Roboterarbeitsplatz stehen, kann es zu keinen Verwechslungen kommen.

5.4.1.1.5 STSrK004: Stelle Produktpalette auf das Transportband:

Anweisung, die Produktpalette zurück auf das Transportband zu stellen. Es werden keine Parameter an Sr übergeben.

STSrK004<Message-ID>0000

Es werden keine Informationen an ST zurückgegeben.

5.4.1.1.6 STSrK005: Tausche Lagerpaletten:

Anweisung, eine Lagerpalette vom Transportband auf einen freien Lagerplatz und die LP des anderen Lagerplatzes zurück aufs Transportband zu stellen. Dieser Befehl ersetzt zwei einzelne „Nimm LP“ und „Gib LP“ Befehle. Als Parameter an Sr wird die LP-Information der neuen LP auf dem Transportband (T) übergeben.

STSrK005<Message-ID>0091{rgybw-}*91

Als Antwort sendet Sr an ST die LP-Information der LP, die von Sr auf das Transportband zurück gestellt wurde.

5.4.1.1.7 STSrK006: Reset/Neustart (Herunterfahren) der Robotersteuerung:

Anweisung, ro von einem definierten Zustand in den Anfangszustand zurückzusetzen. Sr wird dabei heruntergefahren. Es werden keine Parameter an Sr übergeben.

STSrK006<Message-ID>0000

Es werden keine Informationen an ST zurückgegeben.

5.4 Robotersystem

5.4.1.2 Antworten der Robotersteuerung (Sr) an die Steuerung (ST)

5.4.1.2.1 SrSTA001: Befehl wurde verstanden :

Der Befehl wurde empfangen und konnte interpretiert werden.

SrSTA001<Message-ID>0000

5.4.1.2.2 SrSTA002: Befehl wurde ausgeführt :

Der Befehl wurde ohne Fehler ausgeführt.

Je nach Ursprungsbefehl können verschiedene Antworten gesendet werden:

Nach Befehl K002

Nach dem Bestücken wird an ST die abgearbeitete PP-Matrix zurückgegeben.

SrSTA002<Message-ID>0063{rgybw-X}*63

Nach Befehl K003 und K005

Als Antwort sendet Sr an ST die LP-Information der LP, die von Sr auf das Transportband gestellt wurde.

SrSTA002<Message-ID>0091{rgybw-}*91

Alle sonstigen Befehle

SrSTA002<Message-ID>0000

5.4.1.2.3 SrSTF000: Befehl wurde nicht verstanden:

SrSTF000<Message-ID>0000

5.4.1.2.4 SrSTF001: Befehl wurde nicht ausgeführt:

SrSTF001<Message-ID>0000

5.4.1.2.5 SrSTF002: Keine Lagerpalette auf A/B vorhanden:

SrSTF002<Message-ID>0000

5.4.1.2.6 SrSTF003: Keine Produktpalette auf X vorhanden:

SrSTF003<Message-ID>0000

5.4 Robotersystem

5.4.1.2.7 SrSTF004: Plätze A/B sind bereits belegt:

SrSTF004<Message-ID>0000

5.4.1.2.8 SrSTF005: Platz X ist bereits belegt:

SrSTF005<Message-ID>0000

5.4.1.2.9 SrSTF006: Position auf Produktpalette bereits belegt:

SrSTF006<Message-ID>0063{rgybw-}*63

5.4.1.2.10 SrSTF999: Schwerer Fehler ist aufgetreten:

SrSTF999<Message-ID>0000

5.4.1.3 Elementarbefehle der Robotersteuerung (Sr) an das Robotersystem (ro)

5.4.1.3.1 SrroK000: Palette von T nach ABX stellen:

Anweisung, eine Palette vom Transportband auf eine der 3 Positionen auf dem Arbeitsplatz zu legen. Als Parameter an ro kann A, B oder X übergeben werden.

SrroK000<Message-ID>0001{ABX}

Es werden keine Informationen an Sr zurückgegeben.

5.4.1.3.2 SrroK001: Palette von ABX nach T stellen:

Anweisung, eine Palette von einer der 3 Positionen auf dem Arbeitsplatz auf das Transportband zu legen. Als Parameter an ro kann A, B oder X übergeben werden.

SrroK001<Message-ID>0001{ABX}

Es werden keine Informationen an Sr zurückgegeben.

5.4 Robotersystem

5.4.1.3.3 SrroK002: Smartie von Palette und Index nach Palette und Index:

Anweisung, ein Smartie <Index> von der Palette A/B/X auf die Palette A/B/X an die Stelle <Index> zu legen.

SrroK002<Message-ID>0006 {ABX} IIdx {ABX} IIdx

ACHTUNG: IIdx könnte je nach Palette 0-92 (LP) oder 0-62 (PP) sein
Es werden keine Informationen von ro an Sr zurückgegeben.

5.4.1.3.4 SrroK003: Reset (Herunterfahren) des Roboters:

Anweisung, ro in seine Ausgangsposition zurückzufahren. Es werden keine Parameter an ro übergeben.

SrroK003<Message-ID>0000

Es werden keine Informationen an Sr zurückgegeben.

Achtung !!! Der Befehl K003 wird nur mit einem A001 und nicht mit A002 beantwortet. Achtung!
Ob der Roboter den Befehl K003 mit einem alleinigen A001 oder mit A001 und A002 beantwortet, hängt von der Implementierung des Robotersystems ab. Da diese aktuell geändert wird, ist unklar, welches Verhalten gewählt werden wird. Der Simulator erwartet im Moment A001 und A002.

5.4.1.4 Antworten des Robotersystems (ro) an die Robotersteuerung (Sr)

5.4.1.4.1 roSrA001: Elementarbefehl wurde verstanden :

Der Elementarbefehl wurde empfangen und konnte interpretiert werden.

roSrA001<Message-ID>0000

5.4.1.4.2 roSrA002: Elementarbefehl wurde ausgeführt :

Der Elementarbefehl wurde ohne Fehler ausgeführt.

roSrA002<Message-ID>0000

5.4.1.4.3 roSrF000: Elementarbefehl wurde nicht verstanden:

roSrF000<Message-ID>0000

5.4 Robotersystem

5.4.1.4.4 roSrF001: Elementarbefehl wurde nicht ausgeführt:

roSrF001<Message-ID>0000

5.5 Lagersystem

5.5.1 Steuerung und Lagersteuerung

5.5.1.1 Kritische Fehler

Folgende kritische Fehler können von der Lagersteuerung an die Steuerung zurückgegeben werden. Sie können bei allen Aufträgen der Steuerung an das Lagersystem auftreten.

- Unbekannter Befehl / unbekannte Befehlssyntax.

S1STF000<Message-ID>0000

Dieser Fehler würde auftreten, wenn z.B. eine Produkt- anstelle einer Lagerpalette übergeben wird (falsche Matrixgröße).

- Fehler bei lesendem und/oder schreibenden Zugriff auf Datenbasis

Diese Fehlermeldung wird an die Steuerung zurückgegeben, wenn bei einem lesenden und/oder schreibenden Zugriff auf die Bestandsdatei, bzw. auf eine der Log-Dateien ein Fehler auftritt:

S1STF001<Message-ID>0000

5.5.1.2 Befehl- und Antwortverhalten bei den jeweiligen Aufträgen

Im Folgenden werden die Nachrichten zwischen der Lagersteuerung und der Steuerung für die jeweiligen Vorgänge näher spezifiziert. Die in Kapitel ?? beschriebenen Fehlermeldungen können bei jedem Vorgang auftreten und werden bei den einzelnen Vorgängen nicht mehr explizit aufgeführt. Kommandos der Steuerung, die sich nicht an die Befehlssyntax für die jeweiligen Vorgänge halten, werden von der Lagersteuerung ignoriert.

- Anfrage des Bestands einer Farbe Die Fertigungszelle fragt aufgrund einer Gesamtbestandsanfrage von JSAP nacheinander den Bestand jeder im Lager geführten Farbe bei der Lagersteuerung an:

STS1K001<Message-ID>0001{rgybw}

Die Nachricht enthält nur die Farbe, für die der Bestand ermittelt und zurückgemeldet werden soll.

Die Anfrage wird bei erfolgreicher Ausführung mit der ermittelten Anzahl positiv quittiert:
S1STA002<Message-ID>0004xxxx

- Anfrage auf Verfügbarkeit von Smarties einer Farbe

Die Fertigungszelle fragt vor Fertigungsbeginn eines jeden Auftrags nacheinander die Verfügbarkeit der benötigten Anzahl Smarties für jede verwendete Farbe bei der Lagersteuerung an:

STS1K002<Message-ID>0004{rgybw}xxx

Die Nachricht enthält die Farbe und dreistellig die benötigte Menge dieser Farbe.

Bei ausreichendem Bestand wird die Anfrage positiv quittiert:

S1STA002<Message-ID>0000

5.5 Lagersystem

Reicht der Bestand nicht aus, wird die Anfrage mit einem Fehler quittiert, wobei der Steuerung die Anzahl der sich im Lager befindlichen Smarties der angeforderten Farbe mitgeteilt wird:

S1STF002<Message-ID>0003xxx

Die Anzahl der verfügbaren Smarties der Farbe wird dreistellig übertragen.

Des weiteren können bei diesem Kommando folgende Fehler-Nachrichten an die Steuerung gesendet werden:

S1STF003<Message-ID>0000 //Farbe ungültig

S1STF004<Message-ID>0000 //Anzahl ungültig

- Anfrage der Anzahl freier Stellplätze

Die Fertigungszelle fragt vor Start einer Bearbeitungskette, die zu einer Einlagerung einer Lagerpalette führen soll, die Anzahl der noch freien Stellplätze innerhalb des Lagers an:

STS1K003<Message-ID>0000

Die Nachricht enthält keine Nutzdaten.

Die Lagersteuerung antwortet der Steuerung nach erfolgreicher Prüfung des Datenbestands mit folgendem Befehl:

S1STA002<Message-ID>0003xxx

Die Nutzdaten enthalten dreistellig die Anzahl der freien Stellplätze.

- Auftrag zur Einlagerung einer Palette

Die Steuerung kann der Lagersteuerung mit dem Befehl

STS1K004<Message-ID>0094xxx {rgybw-} * 91

den Auftrag zur Einlagerung einer Lagerpalette geben. Die Nutzdaten enthalten eine dreistellige Lagerpaletten-ID und die aktuelle Belegung der Lagerpalette.

Nach erfolgreicher Ausführung wird der Auftrag positiv quittiert:

S1STA002<Message-ID>0000

Konnte die Palette nicht eingelagert werden, erhält die Steuerung eine der folgenden Fehlermeldungen:

S1STF005<Message-ID>0000 //Matrix inhomogen

S1STF006<Message-ID>0000 //kein Lagerplatz frei

S1STF007<Message-ID>0000 //Lagerpalette leer

S1STF008<Message-ID>0000 //mechanischer Fehler beim Einlagern

S1STF009<Message-ID>0000 //Palleten-ID befindet sich bereits im Lager

- Auftrag zur Auslagerung einer Palette

Die Steuerung kann der Lagersteuerung mit dem Befehl

STS1K005<Message-ID>0004 {rgybw} xxx

den Auftrag zur Auslagerung einer Lagerpalette geben. Die Nutzdaten enthalten die Farbe und dreistellig die Anzahl der auszulagernden Smarties.

Die Lagersteuerung quittiert die erfolgreiche Ausführung mit folgender Nachricht:

S1STA002<Message-ID>0094xxx {rgybw-} * 91

5.5 Lagersystem

Die Antwort enthält die dreistellige Lagerpalettennummer und die Belegung der Smarties auf der Palette.

Konnte die Palette nicht ausgelagert werden, erhält die Steuerung eine der folgenden Fehlermeldungen:

S1STF003<Message-ID>0000 //Farbe ungültig

S1STF004<Message-ID>0000 //Anzahl ungültig

S1STF010<Message-ID>0000 //nicht genügend Smarties im Lager

S1STF011<Message-ID>0000 //mechanischer Fehler beim Auslagern

- Herunterfahren des Systems

STS1K006<Message-ID>0000
den Auftrag zum Herunterfahren geben.

- Recovery des Systems

Bricht der Operator im Recovery-Modus beim Hochfahren den letzten, nicht abgeschossenen Befehl ab, wird folgende Fehlermeldung an die Steuerung gesendet:

S1STF999<Message-ID>0000

5.5.2 Lagersteuerung und Lager

5.5.2.1 Kritischer Fehler

Folgender kritischer Fehler kann vom Lager an die Lagersteuerung zurückgegeben werden. Er kann sowohl beim Ein-, als auch beim Auslagerungsvorgang auftreten.

- Unbekannter Elementarbefehl / unbekannte Befehlssyntax

laS1F000<Message-ID>0000

5.5.2.2 Befehl- und Antwortverhalten bei den jeweiligen Aufträgen

Im Folgenden werden die Nachrichten zwischen dem Lager und der Lagersteuerung für den Ein- bzw. Auslagerungsvorgang näher spezifiziert (Elementarbefehle). Neben den hier aufgeführten Nachrichten kann bei jedem Vorgang zusätzlich die in Kapitel ?? beschriebene Nachricht für einen Kommunikationsfehler vom Lager an die Lagersteuerung zurückgegeben werden.

- Einlagerung Palette in Fach X

Dieser Elementarbefehl von der Lagersteuerung veranlasst das Palettenlager, eine Palette, welche vor dem Lager steht, in das dreistellig angegebene Fach zu transportieren:

S1laK001<Message-ID>0003xxx

Nach erfolgreicher Ausführung wird der Elementarbefehl positiv quittiert:

laS1A002<Message-ID>0000

Falls ein Fehler auftritt wird der Elementarbefehl negativ quittiert:

laS1F001<Message-ID>0000

5.5 Lagersystem

Die Lagersteuerung meldet der Steuerung anschließend einen mechanischen Fehler während des Einlagerungsvorgangs.

- Auslagern Palette aus Fach X

Dieser Elementarbefehl von der Lagersteuerung veranlasst das Palettenlager eine Palette aus dem dreistellig angegeben Fach X, auf den vor dem Lager bereitgestellten Schlitten zu transportieren:

S1laK002<Message-ID>0003xxx

Nach erfolgreicher Ausführung wird der Elementarbefehl positiv quittiert:

1aS1A002<Message-ID>0000

Falls ein Fehler auftritt wird der Elementarbefehl negativ quittiert:

1aS1F002<Message-ID>0000

Die Lagersteuerung meldet der Steuerung anschließend einen mechanischen Fehler während des Auslagerungsvorgangs.

5.5.2.3 Austausch TCPIP Kommunikationsschicht Lager

Im Laufe der Analyse der Aufgabenstellung wurde beschlossen, ein einheitliches Modul zur Port-Socket-Kommunikation zu erstellen, welches in allen sendenden und empfangenden Teilsystemen integriert werden soll. Dies betrifft die Lagersteuerung, den Kommandosimulator für Lagersteuerung und Lager und das Lager selbst. Die Kommunikation des übernommenen Softwarestands ist in drei Schichten aufgebaut. Die unterste Schicht "TCPIP.c" von BECK stellt die an Posix angelehnten TCP/IP-Kommunikationsaufrufe zur Verfügung. Die mittlere Schicht "Talk_tcp.c" kapselt die hardwarenahen Aufrufe zu Funktionen zum Aufbau einer Verbindung, Senden und Empfangen von ASCII-Strings und zum Abbau der Verbindung. Die oberste Schicht "New_ethernet.c" enthält alle Funktionen der Netzwerk-Kommunikation. Dies beinhaltet die Überwachung der Port-Socket-Verbindung, das Parsen der empfangenen ASCII-Strings über die "cmd_pars.c" und das Senden der entsprechenden Antwort-Strings.

5.5 Lagersystem

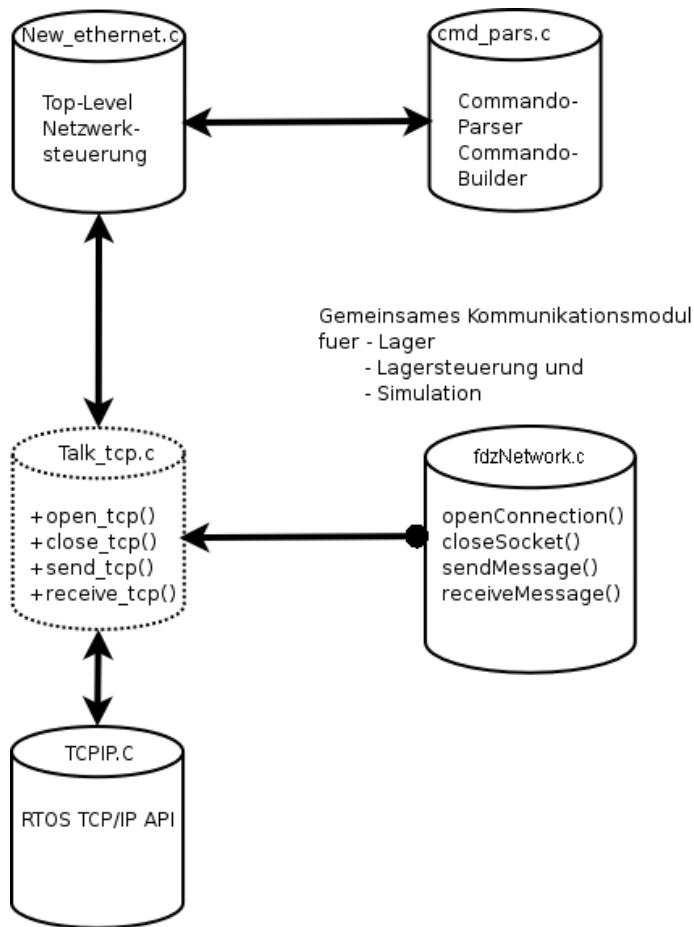


Abbildung 5.1: Anpassung des Netzwerkcodes

Das einheitliche Modul muss die vom Standard abweichenden Aufrufe für den Lagerchip erkennen und entsprechend behandeln. Die chipspezifischen Aufrufe zu Kommunikation der "Talk.tcp.c" sollen durch das einheitliche Modul zur Port-Socket-Kommunikation "fdzNetwork.c" ersetzt werden. Die oberste Schicht zur Verwaltung der Kommunikation "New_ethernet.c" muss im Zusammenhang mit der Einführung des neuen Moduls entsprechend angepasst werden.

5.5.2.4 Einsatz Kommandoparser

In der Spezifikation WS2005/2006 wurde ein neues Protokoll vereinbart, dass mit dem übernommenen Softwarestand nicht überein stimmt. Bei der Besprechung aller änderungen für das Lagersystem wurde festgelegt, dass es einen einheitlichen Kommandoparser "fdzMessageHandler" zur Auswertung empfangener Nachrichten geben soll, welcher in jedem Subsystem eingesetzt wird. Das Modul muss in den bestehenden Code nach dem Empfang einer Nachricht integriert werden. Da die Dateien "cmd_pars.c" und "cmd_pars.h" ausschließlich für die Analyse von Nachrichten des alten Protokolls verantwortlich sind, können diese entfernt werden.

5.5 Lagersystem

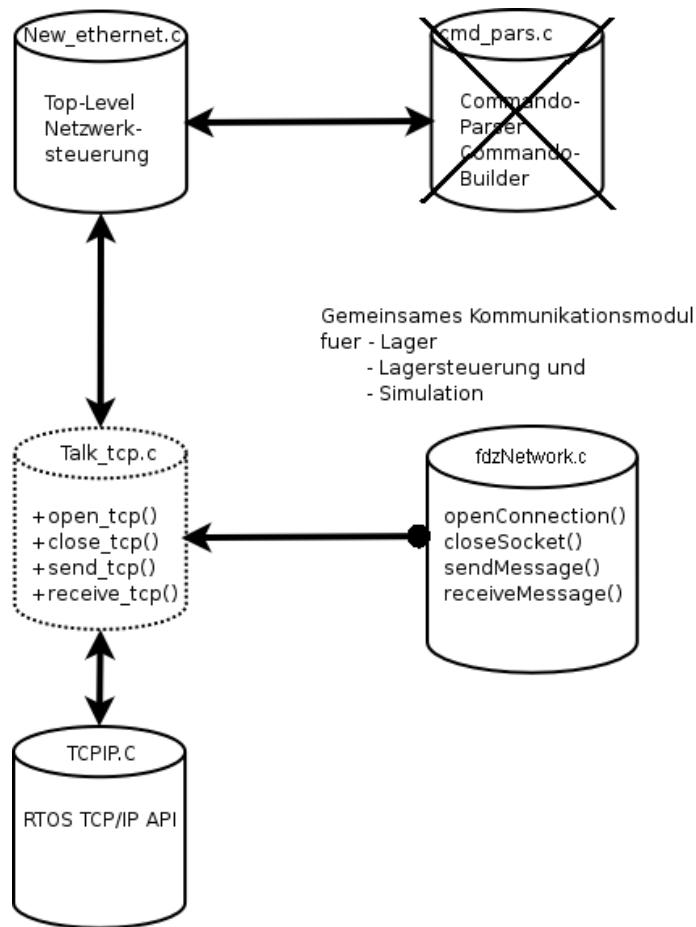


Abbildung 5.2: Anpassung des Netzwerkcodes

5.5.2.5 Implementierung Kommandogenerator

Da es keinen einheitlichen Kommandogenerator für den Aufbau von zu versendenden Nachrichten geben wird, muss dieser für die Generierung der Antworten A001, A002, F000, F001, F002 selbst implementiert werden.

5.6 Transportsystem

5.6.1 Befehl- und Antwortliste

5.6.1.1 Befehle der Steuerung an die Bandsteuerung

- Leeren Schlitten anfordern:

Dieses Kommando wird von Steuerung an Bandsteuerung geschickt, um einen unbeladenen Schlitten an einer Station anzufordern.

STStK001<MessageId>0002<Position>

drei mögliche Werte für <Position> (2 Byte):

'ro' für Roboter
'la' für Lager
'ea' für E/A-Station

- Schlitten freigeben:

Dieses Kommando wird von Steuerung geschickt, um einen Schlitten freizugeben, der nicht mehr benötigt wird. Dieser kann anschließend wieder als leerer Schlitten angefordert werden.

STStK002<MessageId>0002<SchlittenId>

<SchlittenId> kann positive ganzzahlige Werte zwischen 0 und 99 annehmen (2 Bytes).

- Schlitten positionieren:

Dieses Kommando von Steuerung bewirkt, dass ein (möglicherweise bereits beladener) Schlitten an eine bestimmte Station gefahren wird.

STStK003<MessageId>0004<SchlittenId><Position>

Schlitten <SchlittenId> wird an Position <Position> gefahren

<SchlittenId> kann positive ganzzahlige Werte zwischen 0 und 99 annehmen (2 Bytes).

drei mögliche Werte für <Position> (2 Byte):

'ro' für Roboter
'la' für Lager
'ea' für E/A-Station

- Transportsystem herunterfahren:

Mit diesem Kommando weist Steuerung die Bandsteuerung an, das Transportsystem kontrolliert herunterzufahren.

STStK004<MessageId>0000

5.6 Transportsystem

5.6.1.2 Befehle der Bandsteuerung an das Transportband

- Weiche stellen:

Mit diesem Kommando kann Bandsteuerung die Weichen des Transportbandes für eine bestimmte Schlitten Id stellen, um beispielsweise eine Wegplanung zu implementieren.

SttrK001<MessageId>0004<WeichenId><WeichenStellung><SchlittenId>

Für <WeichenId> sind Werte zwischen 0 und 9 gültig (1 Byte), wobei nur 0 und 1 derzeit verwendet werden (da nur zwei Weichen vorhanden sind). <SchlittenId> kann positive ganzzahlige Werte zwischen 0 und 99 annehmen (2 Bytes).

Zwei mögliche Werte für <WeichenStellung> (1 Byte):

- 0 Ausschleusen auf das Hauptband
- 1 Ausschleusen auf das Nebenband

- Fixierstation aktivieren:

Bandsteuerung kann dem Transportband den Befehl schicken, eine Fixierstation zu aktivieren oder zu deaktivieren, dabei wird ein Schlitten mit einer bestimmten Id an einer der Stationen fixiert. Um den nächsten freien Schlitten zu fixieren, muss die Id auf -1 gesetzt werden.

SttrK002<MessageId>0004<FixierstId><FixierstStellung><SchlittenId>

<FixierstId> kann positive ganzzahlige Werte zwischen 0 und 9 annehmen (1 Byte), wobei nur 0 bis 2 derzeit verwendet werden (da nur drei Fixierstationen vorhanden sind).

- 0 EA
- 1 Lager
- 2 Roboter

<SchlittenId> kann positive ganzzahlige Werte zwischen 0 und 99 annehmen (2 Bytes) um einen bestimmten Schlitten zu fixieren. Die Schlitten Id -1 veranlasst das Transportband den nächste freie Schlitten zu fixieren und eine neue Schlitten Id als Antwort zurück zu senden.

Zwei mögliche Werte für <FixierstationStellung> (1 Byte):

- 0 inaktiv
- 1 aktiv

- Transportsystem herunterfahren:

Mit diesem Kommando weist Steuerung die Bandsteuerung an, das Transportsystem kontrolliert herunterzufahren.

SttrK003<MessageId>0000

5.6 Transportsystem

5.6.1.3 Antworten vom Transportband an die Bandsteuerung

Im folgenden wird beschrieben, wie die Kommunikation vom Transportband und Bandsteuerung aussehen könnte.

- Befehl wurde verstanden:

Der Befehl wurde empfangen und konnte interpretiert werden.

trStA001<MessageId>0000

- Befehl wurde ausgeführt:

Der Befehl wurde ohne Fehler ausgeführt.

Nach Befehl K002 mit Schlitten-ID -1:

trStA002<MessageId>0002<SchlittenId>

Alle sonstigen Befehle:

trStA002<MessageId>0000

<SchlittenId> bestimmt, für welchen Schlitten das letzte Kommando durchgeführt wurde.
<SchlittenId> kann positive ganzzahlige Werte zwischen 0 und 99 annehmen (2 Bytes)

- Befehl wurde nicht verstanden:

trStF000<MessageId>0000

- Befehl wurde nicht ausgeführt:

trStF001<MessageId>0001<FehlerId>

<FehlerId> kann die folgenden Werte annehmen (1 Byte):

- 0 - Steuerung aus (SPS funktioniert nicht)
- 1 - kein Druck vorhanden (Weichenstellung funktioniert nicht)
- 2 - Sicherungsfall
- 3 - Antrieb Band 1 Störung
- 4 - Antrieb Band 2 Störung
- 5 - Antrieb Band 3 Störung
- 6 - Not-Aus betätigt
- 7 - allgemeiner / nicht spezifizierter Fehler

5.6 Transportsystem

5.6.1.4 Antworten der Bandsteuerung an die Steuerung

- Befehl wurde verstanden:

Der Befehl wurde empfangen und konnte interpretiert werden.

StSTA001<MessageId>0000

- Befehl wurde ausgeführt:

Der Befehl wurde ohne Fehler ausgeführt.

Je nach Ursprungsbefehl können verschiedene Antworten gesendet werden:

Nach Befehl K001:

Bandsteuerung gibt die ID des angeforderten leeren Schlittens zurück.

StSTA002<MessageId>0002<SchlittenId>

<SchlittenId> kann positive ganzzahlige Werte zwischen 0 und 99 annehmen (2 Bytes).

Alle sonstigen Befehle:

StSTA002<MessageId>0000

- Befehl wurde nicht verstanden:

StSTF000<MessageId>0000

- Befehl wurde nicht ausgeführt:

StSTF001<MessageId>0000

- Schlitten ist nicht angekommen:

Ein an einer bestimmten Position angeforderter Schlitten kommt nicht an.

StSTF002<MessageId>0002<SchlittenId>

<SchlittenId> kann positive ganzzahlige Werte zwischen 0 und 99 annehmen (2 Bytes).

- Allgemeiner Hardware-Fehler:

Im Transportsystem ist ein Hardware-Fehler aufgetreten. Die genaue Art des Fehlers wird als Fehler-Id mitgeliefert.

StSTF003<MessageId>0003<FehlerId>

5.6 Transportsystem

<FehlerId> kann die folgenden Werte annehmen (3 Byte):

- 0 - Steuerung aus (SPS funktioniert nicht)
- 1 - kein Druck vorhanden (Weichenstellung funktioniert nicht)
- 2 - Sicherungsfall
- 3 - Antrieb Band 1 Störung
- 4 - Antrieb Band 2 Störung
- 5 - Antrieb Band 3 Störung
- 6 - Not-Aus betätigt

- Das Transportsystem kann nicht weiterarbeiten:

Das Transportsystem kann nicht weiterarbeiten, wenn vom Transportband der Fehler F001 mit FehlerId 7 empfangen wurde.

St STF 999<MessageId>0000

5.7 Eingabe/Ausgabe-Station

5.7.1 Befehl- und Antwortliste

5.7.1.1 Befehle der Steuerung an die E/A-Stationensteuerung

- Leere Lagerpalette ausschleusen:

Die betroffene Lagerpalette wird vom Schlitten genommen und verlässt das System.

STSeK000<MessageId>0003<PalettenId>

- Eine Produktpalette ausschleusen:

Die betroffene Produktpalette wird vom Schlitten genommen und verlässt das System.

STSeK001<MessageId>0063{rgybw-}*63

Als Parameter wird die aus 63 Zeichen bestehende Matrix der bestückten Produktpalette übergeben.

- Lagerpalette mit Bestückung befüllen:

Der Befehl bewirkt, dass die Steuerung E/A-Station eine leere Lagerpalette mit der angegebenen Bestückung füllt.

STSeK002<MessageId>0091{rgybw-}*91

Als Parameter wird die aus 91 Zeichen bestehende Matrix der Bestückung mitgeliefert.

- Lagerpalette mit Paletten-ID einschleusen:

Es wird die vorher gefüllte Lagerpalette mit der ID <PalettenId> in das System eingebracht.

STSeK003<MessageId>0003<PalettenId>

<PalettenId> kann Werte zwischen 0 und 999 annehmen (3 Byte).

- Leere Produktpalette einschleusen:

Dieser Befehl bewirkt, dass eine leere Produktpalette, die für einen neuen Auftrag bestückt werden soll, ins System eingeschleust wird.

STSeK004<MessageId>

- System herunterfahren:

STSeK005<MessageId>0000

- NICHT leere Lagerpalette ausschleusen: Die betroffene Lagerpalette wird vom Schlitten genommen, die restlichen Smarties werden von der Lagerpalette genommen. Die Bestückungs-Matrix der Palette wird zusammen mit der PalettenID in der Bestandsdatei der E/A-Station gespeichert, um in der Folge auf diese zugreifen zu können.

5.7 Eingabe/Ausgabe-Station

STSeK006<MessageId>0094<PalettenId>{rgybw-}*91

Hierbei belegt die PalettenId wieder 3 Bytes und kann positive ganzzahlige Werte von 0 bis 999 annehmen.

5.7.1.2 Antworten der E/A-StationsSteuerung an die Steuerung

- Befehl wurde verstanden:

Der Befehl wurde empfangen und konnte interpretiert werden.

SeSTA001<MessageId>0000

- Befehl wurde ausgeführt:

Der Befehl wurde ohne Fehler ausgeführt:

SeSTA002<MessageId>0000

Je nach Ursprungsbefehl können verschiedene Antworten gesendet werden:

Nach Befehl K002:

Steuerung E/A-Station gibt die ID der befüllten Lagerpalette zurück sowie ihre tatsächliche Bestückung.

SeSTA002<MessageId>0094<PalettenId>{rgybw-}*91

<PalettenId> kann positive ganzzahlige Werte zwischen 0 und 999 annehmen (3 Bytes).

Danach folgt die Bestückung (91 Byte).

Nach Befehl K003:

Steuerung E/A-Station gibt die ID der eingeschleusten Lagerpalette zurück sowie ihre tatsächliche Bestückung.

SeSTA002<MessageId>0094<PalettenId>{rgybw-}*91

<PalettenId> kann positive ganzzahlige Werte zwischen 0 und 999 annehmen (3 Bytes).

Danach folgt die Bestückung (91 Byte).

Alle sonstigen Befehle:

SeSTA002<MessageId>0000

- Befehl wurde nicht verstanden:

SeSTF000<MessageId>0000

- Befehl wurde nicht ausgeführt:

SeSTF001<MessageId>0000

5.7 Eingabe/Ausgabe-Station

- Das E/A-System kann nicht weiterarbeiten:

Das E/A-System kann dann nicht weiterarbeiten, wenn die Bestands- oder Recoverydatei fehlerhaft ist oder eine Exception ausgelöst wird.

StSTF999<MessageId>0000

6 Fehlerfälle

6.1 Fehlerfälle für die Steuerung

Dieser Abschnitt behandelt die möglichen Fehlerfälle, die zwischen JSAP, Steuerung und Subsystemen vorkommen können.

6.1.1 Allgemeine Fehler

Folgende allgemeine Fehler können bei der Kommunikation auftreten:

- Fehler bei syntaktischem Aufbau des Befehls an die Subsysteme
- Fehler bei syntaktischem Aufbau des Befehls an das JSAP
- Kommando/Befehl unbekannt

Folgende grundsätzliche Fehler können bei dem Zugriff auf Daten innerhalb der Steuerung auftreten:

- Fehler bei lesenden und/oder schreibenden Zugriff auf die Queue-Datei
- Fehler bei lesenden und/oder schreibenden Zugriff auf die Log-Datei

Außerdem können jederzeit Systemfehler wie Systemabstürze oder Stromausfälle auftreten. Sie werden daher nicht explizit bei jedem Ablauf aufgelistet.

6.1.2 Fehler zwischen Steuerung und Subsystemen

Subsysteme können sich nicht mehr mit der Steuerung verbinden

- Da die Steuerung als Server für die Subsysteme agiert, weiß sie, welche Subsysteme mit ihr verbunden und betriebsbereit sind (siehe hierzu auch nochmal unter ??). Sollte sich ein Subsystem nicht mehr mit der Steuerung bis zum Ablauf eines Timeouts verbinden lassen, fährt es sich herunter. Fehlt ein Subsystem, schickt die Steuerung an alle anderen Subsysteme den Befehl "herunterfahren". Für die entsprechenden Nachrichtenspezifikationen siehe unter ?? (Robotersystem), ?? (Lagersystem), ?? (Transportsystem) und ?? (E/A-Station).

Meldungen nicht verstanden

6.1 Fehlerfälle für die Steuerung

- Steuerung verschickt einen Befehl an ein Subsystem. Dieser wird mit der Fehlermeldung "Befehl nicht verstanden" vom Subsystem quittiert. Daraufhin schreibt die Steuerung die Fehlermeldung in ihre Log-Datei und verschickt den zuvor gesendeten Befehl mit einem neuen Zeitstempel noch mal. Sollte sich der Fehlerfall 5 mal wiederholen, wird bei der Steuerung eine Nachricht für den Operator generiert und die Subsysteme werden per Befehl heruntergefahren.

Antworten nicht verstanden (ACK1, ACK2, fehlende Smarties am Roboter (xxx))

- Sollte die Steuerung von den Subsystemsteuerungen die Antwort nicht verstanden haben, wird die Steuerung zu der jeweiligen Subsystemsteuerung nach einer gewissen Zeit den Ursprungsbefehl aus der Log-Datei der Steuerung mit dem Ursprungszeitstempel erneut an die Subsystemsteuerung schicken.

Subsystem meldet Fehlerfall (Weiß nicht mehr weiter!)

- Ein Subsystem meldet der Steuerung, dass es nicht mehr weiter weiß. Dies geschieht nur, wenn das Subsystem schon selbst versucht hat, den Fehler zu beheben. Da die Steuerung keine Behebung vornehmen kann, um den Fehler zu korrigieren, wird das gesamte System heruntergefahren. Hierbei sendet die Steuerung an die restlichen Subsysteme den Befehl "herunterfahren". Sind auch das fehlerbehaftete Subsystem und die Steuerung selbst heruntergefahren, kann der Fehler manuell behoben werden.

6.1.2.1 Fehler zwischen JSAP und Steuerung

Verbindung zwischen JSAP und Steuerung gestört

- Bei der Verbindung zwischen JSAP und der Steuerung tritt die Steuerung als Client und JSAP als Server auf. Die Steuerung kommuniziert mit dem JSAP, wenn ein Auftrag angenommen wird.
- **Fall: Verbindung zwischen JSAP und Steuerung bricht ab**
 - Die Steuerung hat keinen Auftrag
Wenn die Steuerung noch keinen Auftrag hat, warten wir und versuchen alle 60 Sekunden eine Verbindung herzustellen. Irgendwann wird dann ein Auftrag von JSAP geschickt. Wenn das nicht funktionieren sollte, muss JSAP eigene Kontrollmechanismen haben.
 - Auftrag vorhanden
Der Auftrag wird zwischenzeitlich normal abgearbeitet. Wenn irgendwann wieder eine Verbindung zu JSAP besteht und wir mit dem Auftrag fertig sind, teilen wir dies JSAP mit („Produktion abgeschlossen, Zeit benötigt, Smartieverbrauch“).
 - Auftrag abgearbeitet
Hierbei hat die Steuerung einen Auftrag abgearbeitet und kann kein "Befehl ausgeführt" schicken. Da TCP/IP von sich aus erkennt, wenn die Verbindung wieder hergestellt ist, versucht TCP/IP selbstständig, "Befehl ausgeführt" nochmal zu senden.

6.1 Fehlerfälle für die Steuerung

- **Fall: Auftrag von JSAP wird von Steuerung nicht verstanden**

Das JSAP verschickt einen Auftrag an die Steuerung. Dieser wird mit der Fehlermeldung "Befehl nicht verstanden" von der Steuerung quittiert. Der empfangene Befehl wird im Logfile zur späteren Fehleranalyse gespeichert.

- **Fall: Steuerung ist bereits beschäftigt**

Versucht JSAP eine Anfrage an Steuerung zu stellen, während die Steuerung einen Auftrag produziert, wird diese Anfrage mit einer Fehlermeldung quittiert. Dieses Verhalten von JSAP wäre nicht konform mit dem definierten Verhalten von JSAP. Mögliche Fälle wären z.B. Lagerstatus wird abgerufen während ein Auftrag produziert wird, oder ein neuer Auftrag wird während der Produktion Steuerung mitgeteilt. Die einzige Ausnahme ist der Befehl "Auftrag abbrechen".

Auftrag annehmen

- **Fall: Zu wenig Smarties**

Nachdem die Steuerung das Lager geprüft und festgestellt hat, dass es zu wenig Smarties gibt, schickt sie dem JSAP die Fehlermeldung, dass der Auftrag nicht ausgeführt werden kann.

Dieser Fehlerfall sollte nicht vorkommen, da JSAP die Möglichkeit hat den Lagerstatus der FDZ abzufragen.

- **Fall: Ein Auftrag wird bereits produziert**

Siehe Fall "Steuerung ist bereits beschäftigt"

Auftrag abbrechen

- **Fall: Kein Auftrag vorhanden**

Möchte JSAP einen Auftrag abbrechen, Steuerung hat aber zu diesem Zeitpunkt keinen Auftrag, wird dies JSAP mitgeteilt.

- **Fall: Timestamp des Auftrages entspricht nicht aktuellen Auftrag**

JSAP versucht einen Auftrag abzubrechen, der übermittelte Timestamp passt allerdings nicht zum aktuellen Auftrag. JSAP empfängt in diesem Fall eine Fehlermeldung, dass der Auftrag nicht abgebrochen wurde, da der Timestamp falsch ist.

Lagerstatus abrufen

- **Fall: JSAP versucht während Produktion eines Auftrages den Lagerstatus abzurufen.**

Siehe Fall "Steuerung ist bereits beschäftigt"

6.1.2.2 Fehler in der Steuerung

Recovery Log-Datei nicht lesbar

- Sollte die Recovery Log-Datei nach einem fehlerbedingten Neustart für die Steuerung nicht mehr lesbar sein, bricht die Steuerung den Neustart ab und gibt dem Operator eine Meldung, das Recovery-Log von Hand zu löschen. Damit wird ein eventuell angefangener Auftrag nicht weiter bearbeitet. Wurde die Recovery-Log-Datei von Hand gelöscht, kann die Steuerung wieder gestartet werden. Ist der Startvorgang abgeschlossen, meldet die

6.1 Fehlerfälle für die Steuerung

Steuerung dem JSAP ihre Bereitschaft. Da die Steuerung nicht mit einem vorgesehenen ACK2 den Auftrag quittiert hat, weiß das JSAP, dass der Auftrag nicht ausgeführt wurde und schickt den gleichen Auftrag nochmals an die Steuerung.

6.1.2.3 Schwerwiegender Fehler 999

Der schwerwiegende Fehler 999 wird von einem Subsystem an die Steuerung geschickt, wenn das Subsystem nicht mehr in der Lage ist, einen Fehlerfall selbst zu beheben. Dieser Fehler veranlasst die Steuerung, alle übrigen Subsysteme kontrolliert herunter zu fahren.

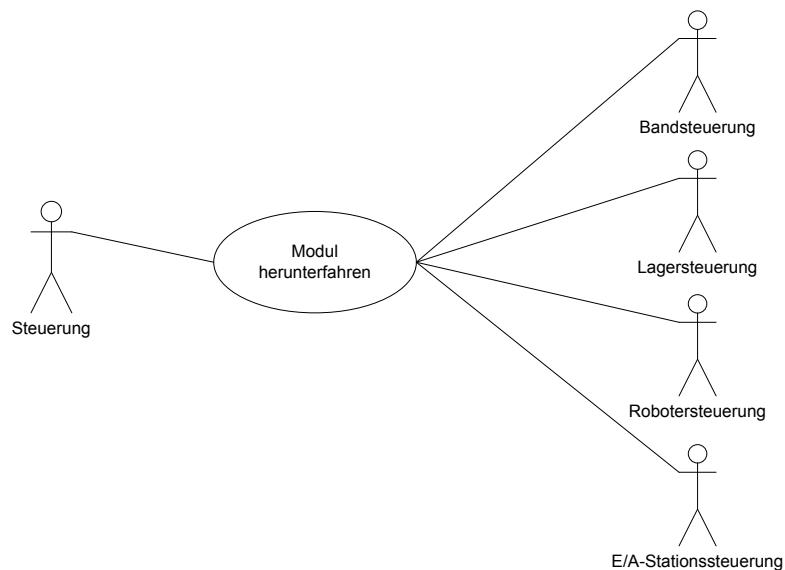


Abbildung 6.1: Verhalten bei schwerwiegendem Fehler

6.2 Fehlerfälle für das Robotersystem

6.2.1 Kommunikationsfehler

Bei der Kommunikation des Robotersystems werden Befehle von Steuerung an Robotersteuerung und Elementarbefehle von Robotersteuerung an Roboter geschickt. Da die im folgenden Text beschriebenen Fehler bei Befehlen und Elementarbefehlen auftreten können, wird der Einfachheit halber immer von Befehlen gesprochen. Bei der Kommunikation können folgende Kommunikationsfehler auftreten:

- Befehle können wegen syntaktischen Fehler nicht verarbeitet werden z.B. Befehlsnummer passt nicht mit Länge der Nutzdaten zusammen, Befehlsnummer passt nicht mit Art der Nutzdaten zusammen (falsche Buchstaben im Payload), ...
- Befehlsnummer wird vom Empfänger nicht erkannt, evtl wurde der Befehl an das falsche Subsystem geschickt.
- Befehle kommen aufgrund eines Netzwerkfehlers nur teilweise oder verstümmelt an (ist von den ersten beiden Fällen nicht unbedingt zu unterscheiden, läuft aber auf die selbe Fehlermeldung hinaus). Dies sollte bei TCP/IP nicht vorkommen, soll hier aber trotzdem als theoretischer Fehler angeführt werden.

In diesen Fällen wird statt der ACK1 (A001) Antwort eine Fehlermeldung (F000) an das entsprechende System geschickt. Gegebenenfalls kann der Befehl dann erneut gesendet werden.

6.2.2 Fehler, die von der Robotersteuerung erkannt werden

- ein Befehl kann aus logischen Gründen nicht ausgeführt werden. Dies tritt auf, wenn z.B. eine Palette aufs Band gelegt werden soll, obwohl keine Palette auf dem Robotertisch bereitsteht, also vorher keine entsprechende Palette auf den Robotertisch gestellt wurde (analog beim Abholen von Paletten vom Band, wenn entsprechende Palette beim Roboter schon vorhanden ist).

In diesem Fall sendet Robotersteuerung die dem Fehler entsprechende Fehlermeldung an Steuerung.

Liegt ein mechanischer Fehler vor¹, so schickt Robotersteuerung die Fehlermeldung F001. Ebenso schickt Robotersteuerung F001, falls der Roboter den gesendeten Befehl mit einer Fehlermeldung quittiert (unabhängig von einem mechanischen Problem, sondern auch z.B. falls der Roboter einen Befehl von Robotersteuerung nicht verstanden hat).²

Robotersteuerung kann nicht überprüfen, ob die von Control übermittelte Matrix der Lagerpalette korrekt ist (d.h. ob die Matrix der reellen Belegung der LP entspricht). Hierzu wäre eine Aufrüstung mit einer Kamera nötig.

¹ Wie ein solcher mechanischer Fehler erkennbar ist, hängt vom Roboter und der Implementierung der Robotersoftware ab. Da aktuell ein neuer Roboter installiert wird und dessen Software dementsprechend neu geschrieben werden muss, ist noch nicht klar, inwiefern solche Fehler erkannt werden können.

² Hier wäre in Zukunft eine differenziertere Fehlerbehandlung denkbar. So könnte z.B. im Fall von F000 seitens ro Sr versuchen, den Befehl noch 3x zu senden, und dann erst F001 an ST zu senden.

6.2 Fehlerfälle für das Robotersystem

6.2.3 Kritische Fehler des Robotersystems

Neben den oben genannten Fehlern können am Roboter so genannte „kritische Fehler“ auftreten. Bei einem kritischen Fehler des Roboters muss die gesamte FdZ anhalten und es ist ein manueller Eingriff eines Mitarbeiters/Operators nötig, um z.B. den Roboter neu zu justieren. Voraussetzung für die Behandlung eines kritischen Fehlers ist, dass der Roboter nach einem solchen aufhört zu arbeiten/sich zu bewegen. Sollte der Roboterarm z.B. mit dem Lager kollidieren und danach aber normal weiterarbeiten, gibt es für die Software keine Möglichkeit diesen kritischen Fehler festzustellen. Kritische Fehler können z.B. sein

- Roboterarm hat sich verkantet (mit sich selbst)
- Pneumatik-Schläuche des Roboterarms sind abgerissen (z.B. bei zu weiter Umdrehung um die eigene Achse)
- Roboterarm ist mit einem physischen Gegenstand (Lager, Band, Wand, Mensch) kollidiert
- Hardware-Defekt des Roboters

Diese kritischen Fehler können von der Software nicht unterscheidbar erkannt werden und machen sich nur durch das Nicht-Reagieren des Roboters bemerkbar³. Darum wird, wenn der Roboter nicht mehr reagiert, ein F001 von Robotersteuerung an Steuerung gesendet, worauf hin Steuerung die gesamte FdZ anhalten soll. Bevor das System wieder hochgefahren werden kann, muss ein Mitarbeiter den Roboter inspizieren und Maßnahmen ergreifen, um den kritischen Fehler zu beheben (Roboter neu justieren, Schläuche wieder anbringen, Notarzt rufen, ...). Der laufende Auftrag am Roboter ist damit verloren: Entweder muss der Mitarbeiter alle Paletten entfernen und zur E/A Station geben, oder Steuerung leer den Arbeitsplatz⁴.

6.2.4 Roboterfehler, während KEIN Auftrag abgearbeitet wird

Wenn dieser Fall auftritt, wird zunächst der Steuerung kein Fehler gemeldet. Erst zu dem Zeitpunkt, an welchem der nächste Befehl an die Robotersteuerung gesendet wird, wird dieser Fehler registriert, zurückgemeldet und von der Steuerung bearbeitet.

6.2.5 Fehler in der Netzwerkverbindung

Falls es zu einem Netzwerkfehler während der Abarbeitung eines Befehls kommt, kann dieser nicht automatisch vom System behoben werden. Die Steuerung wartet währenddessen auf die Rückmeldung des Subsystems bis die Verbindung wiederhergestellt wurde. Es kann dabei nötig sein, dass ein Operator benachrichtigt wird, welcher den Fehler beheben muss.

³Wie ein solcher mechanischer Fehler erkennbar ist, hängt vom Roboter und der Implementierung der Robotersoftware ab. Da aktuell ein neuer Roboter installiert wird und dessen Software dementsprechend neu geschrieben werden muss, ist noch nicht klar, inwiefern solche Fehler erkannt werden können.

⁴Die aktuelle Implementierung von Control sieht ein Leeren des Arbeitsplatzes im Fehlerfall nicht vor, d.h. es sind manuelle Eingriffe nötig.

6.3 Fehlerfälle für das Lagersystem

Um Missverständnissen vorzubeugen erfolgt hier kurz eine genaue Beschreibung, was als ein Fehler im Gegensatz zum Recovery verstanden wird. Der Begriff der Fehlerbehandlung bezieht sich auf die Behandlung aller Fehlerfälle im aktiven Betrieb, die keinen Stromausfall als Ursache haben. Dazu gehören Fehler bei der Kommunikation, logische Fehler bei der Auswertung von Befehlen an und von der Lagersteuerung, Not-Aus Betätigung und mechanische Fehler bei der Ausführung einer Ein- bzw. Auslagerung. Bei der Analyse des übernommenen Softwarestands zur Steuerung der Lager-Hardware ist aufgefallen, dass dort die Fehlerbehandlung bereits implementiert ist. Bei der Besprechung aller nötigen Änderungen wurde beschlossen, diese Funktionen hardwarenah zu belassen.

In diesem Kapitel wird zu jedem Ablauf auf möglich auftretende Fehler eingegangen.

6.3.1 Allgemeine Fehler

Folgende grundsätzliche bzw. allgemeine Fehler können bei der Kommunikation auftreten:

- Fehler bei syntaktischem Aufbau des Befehls an die Lagersteuerung
- Fehler bei syntaktischem Aufbau des Elementarbefehl an das Lager
- Kommando/Befehl unbekannt

Folgende grundsätzliche Fehler können bei dem Zugriff auf Daten innerhalb des Lagersystems auftreten:

- Fehler bei lesendem und/oder schreibendem Zugriff auf den Lagerbestand in der Lagerbestandsdatei
- Fehler bei lesendem und/oder schreibendem Zugriff auf die zwischengespeicherten Befehle in den Log-Dateien sowohl bei Lagersteuerung als auch Lager

Bei Auftreten dieser schwerwiegenden Schreib-/Lese-Fehler bei der Lagersteuerung, kann das Lagersystem nicht mehr weiterarbeiten. Dies muss der Steuerung nach dem nächsten, von ihr erhaltenen Kommando, mitgeteilt werden. (Siehe Abschnitt ??) Sollte ein Schreib-/Lese-Fehler beim Lager auftreten, muss es sofort heruntergefahren werden. Der Operator muss den Fehler am LCD erkennen und kann den Fehler z.B. durch Austausch des Lagerchips beheben. Nach Beheben des Fehlers und Neustart des Lagers kommt die Recoverybearbeitung zum Tragen.

Außerdem können jederzeit Systemfehler wie Systemabstürze oder Stromausfälle auftreten. Sie werden daher nicht explizit bei jedem Ablauf aufgelistet. Allerdings wurde in den Ablaufbeschreibungen auf Maßnahmen verwiesen, die einen Stillstand des Systems nach Auftreten eines solchen Fehlers vermeiden sollen, d.h. die Möglichkeit zu einem Wiederanlauf geben sollen. Die Maßnahmen bzw. Reaktionen zum Wiederanlauf des Auftrags nach solchen und den folgenden Fehlern werden detailliert im Kapitel "Recovery" erläutert.

6.3 Fehlerfälle für das Lagersystem

6.3.2 Fehler bei Hochfahren und Anmelden am System

Bei diesem Vorgang können sowohl in der Lagersteuerung als auch im Lager nur allgemeine Fehler auftreten. Diese sind im vorangegangenen Unterkapitel näher erläutert.

6.3.3 Fehler bei Anfrage "Bestand einer Farbe"

Bei diesem Vorgang innerhalb der Lagersteuerung können nur allgemeine Fehler auftreten. Diese sind im vorangegangenen Unterkapitel näher erläutert.

6.3.4 Fehler bei Anfrage "Verfügbarkeit einer Farbe"

Bei diesem Vorgang innerhalb der Lagersteuerung tritt ein logischer Fehler auf, wenn sich im Lager weniger Smarties einer bestimmten Farbe befinden, als die Steuerung angefragt hat. Hintergrund ist, dass das JSAP-System nur bestandstechnisch erfüllbare Aufträge an die Steuerung übertragen darf.

6.3.5 Fehler bei Anfrage "Anzahl freien Stellplätze im Lager"

Bei diesem Vorgang innerhalb der Lagersteuerung können nur allgemeine Fehler auftreten. Diese sind im vorangegangenen Unterkapitel näher erläutert.

6.3.6 Fehler bei Auftrag zur Einlagerung einer Palette

Die folgenden Fehler können neben den allgemeinen Fehlern bei der Abarbeitung eines Einlagerauftrags auftreten.

logischer Fehler

- Ein Auftrag zur Einlagerung einer leeren Lagerpalette wurde an die Lagersteuerung übertragen. Dies ist aufgrund der Vereinbarungen am Anfang des Projekts nicht erlaubt.
- Auftrag zur Einlagerung einer Produktpalette wurde an die Lagersteuerung übertragen. Dies ist nicht erlaubt.
- Es ist in der Bestandsverwaltung kein freier Stellplatz mehr für die Einlagerung der Lagerpalette vorhanden. Dies darf nicht passieren. Die Steuerung hätte vorher anfragen müssen, ob noch Plätze frei sind.
- Die Lagersteuerung hat einen Auftrag zur Einlagerung einer Palette in ein belegtes Fach an die Lagerhardware übermittelt. Dies muss erkannt und mit einem Fehler quittiert werden.

mechanischer Fehler

6.3 Fehlerfälle für das Lagersystem

- Bei der physikalischen Ein- oder Auslagerung einer Palette meldet die Mechanik aufgrund defekter Hardware oder einer verklemmten Palette keinen (!) Fehler an die Lagersteuerung zurück, sondern fährt sich nach Rückfrage beim Operator herunter. Nach Beheben des Fehlers und Neustart des Lagers tritt der Recovery-Fall ein (siehe Abschnitt ??, so daß der zuletzt aktive Auftrag weiter bearbeitet werden kann).

6.3.7 Fehler bei Auftrag zur Auslagerung einer Palette

Die folgenden Fehler können neben den allgemeinen Fehlern bei der Abarbeitung eines Auslagerauftrags auftreten. Zu jedem aufgetretenen Fehler wird eine entsprechende Nachricht an die Steuerung geschickt .

logischer Fehler

- Im Lager befinden sich nicht genügend Smarties einer bestimmten Farbe. Die Steuerung hätte vor Auftragsbeginn anfragen müssen, ob ausreichend Smarties dieser Farbe vorhanden sind.
- Die Lagersteuerung hat einen Auftrag zur Auslagerung einer Palette aus einem leeren Fach an die Lagerhardware übermittelt. Dies muss erkannt und mit einem Fehler quittiert werden.

mechanischer Fehler

- Bei der physikalischen Ein- oder Auslagerung einer Palette meldet die Mechanik aufgrund defekter Hardware oder einer verklemmten Palette keinen (!) Fehler an die Lagersteuerung zurück, sondern fährt sich nach Rückfrage beim Operator herunter. Nach Beheben des Fehlers und Neustart des Lagers tritt der Recovery-Fall ein (siehe Abschnitt ??, so daß der zuletzt aktive Auftrag weiter bearbeitet werden kann).

6.3.8 Fehler bei Auftrag zum Herunterfahren des Systems

Bei diesem Vorgang können in der Lagersteuerung nur allgemeine Fehler auftreten. Diese sind im vorangegangenen Unterkapitel näher erläutert.

6.3.9 Fehler durch Recovery-Fall während des Systemstarts

Wird beim Hochfahren der Lagerhardware festgestellt, dass ein Auftrag nicht vollständig abgeschlossen wurde, tritt der Recovery-Fall ein. D.h. es wird dem Lageroperator zur Wahl gestellt, den letzten empfangenen Befehl der Lagersteuerung komplett zu wiederholen. Lehnt er die Wiederholung ab, wird ein dem Auftrag (Ein-, Auslagerung) entsprechender Fehler an die Lagersteuerung gemeldet.

6.3 Fehlerfälle für das Lagersystem

6.3.10 Fehler durch Verbindungsabbruch zur Steuerung

Bricht im Produktivbetrieb die Netzwerkverbindung zwischen Steuerung und Lagersteuerung zusammen, wird das gesamte System solange angehalten, bis ein zuständiger Operator das Problem behebt.

6.4 Fehlerfälle für das Transportsystem

Während der Arbeit mit dem System kann es auch zu "ungeahnten" Zwischenfällen kommen. Diese gehen größtenteils von der Hardware aus. Da in solchen Fällen softwareseitig meist nicht viel ausgerichtet werden kann, werden Fehlermeldungen generiert und zunächst auf dem PC der Bandsteuerung ausgegeben. Hierbei hat ein Mitarbeiter dann die Möglichkeit, den Befehl wiederholen zu lassen oder abzubrechen. Treten auch beim wiederholten Mal Probleme auf oder wählt der Mitarbeiter "Abbrechen" aus, wird eine Meldung an Steuerung gesendet. Manche Fehler erfordern, dass sofort eine Meldung an Steuerung geschickt wird.

Im folgenden Abschnitt werden die möglicherweise auftretenden Fehler spezifiziert.

6.4.1 Bandsteuerungs-Fehler 1: Befehl nicht verstanden

Beschreibung:

Ein von Steuerung gesendetes Kommando kann nicht identifiziert und damit nicht ausgeführt werden.

6.4.2 Bandsteuerungs-Fehler 2: Befehl nicht ausgeführt

Beschreibung:

Ein von Steuerung gesendetes Kommando konnte nicht ausgeführt werden.

Folgen:

Es wird eine Fehlermeldung an Steuerung gesendet.

6.4.3 Bandsteuerungs-Fehler 3: Schlitten kommt nicht an

Beschreibung:

Der Schlitten kommt nach einer maximalen Zeit nicht - wie erwartet - am Zielsensor an bzw. sein Ankommen wird nicht erkannt.

Ursachen:

- Jemand nimmt den Schlitten vom Band
- Schlitten bleibt hängen
- Sensor defekt
- Sperre funktioniert nicht

6.4 Fehlerfälle für das Transportsystem

Folgen:

Es erfolgt zunächst die Ausgabe einer Fehlermeldung am Bandsteuerungs-PC für den Mitarbeiter. Zusammen mit dieser Meldung wird die letzte von Bandsteuerung registrierte Position des Schlittens ausgegeben. Der Operator muss diesen bei einem Retry wieder auf diese Position setzen. Beim wiederholten Auftreten oder bei Abbruch wird eine Fehlermeldung an Steuerung gesendet.

6.4.4 Bandsteuerungs-Fehler 4: Allgemeiner Hardware-Fehler im Transportsystem

Andere Fehler, die erkannt werden können.

- ID 0 Steuerung aus
- ID 1 kein Druck vorhanden
- ID 2 Sicherungsfall
- ID 3 Antrieb Band 1 Störung
- ID 4 Antrieb Band 2 Störung
- ID 5 Antrieb Band 3 Störung
- ID 6 Not-Aus betätigt
- ID 7 Timeout
- ID 8 kein freier Schlitten mehr vorhanden

1. ID 0 Steuerung aus

Beschreibung: Die Steuerung ist außer Betrieb.

Ursachen:

- SPS ausgeschaltet
- Stromzufuhr unterbrochen

2. ID 1 kein Druck vorhanden

Beschreibung: Weichenstellung bzw. Stopper funktionieren nicht.

Ursachen:

- Schlauchverbindung abgerissen
- Drucklufterzeugung defekt

3. ID 2 Sicherungsfall

Beschreibung: Sicherungsfall

Ursachen:

6.4 Fehlerfälle für das Transportsystem

- Stromaufnahme zu hoch
- Kurzschluss

4. ID 3 Antrieb Band 1 Störung

5. ID 4 Antrieb Band 2 Störung

6. ID 5 Antrieb Band 3 Störung

Beschreibung: Motorschutzschalter fällt

Ursachen:

- Fehler an Motor 1,2 oder 3.
- Stromaufnahme des Motors zu hoch.(Fehler in Motorwicklungen bzw. Last am Motor zu hoch usw.)

7. ID 6 Not-Aus betätigt

Beschreibung: Not-Aus Knopf wurde betätigt

8. ID 7 Timeout

Beschreibung: Kommt ein Schlitten bei verlassen eines Sensors nicht innerhalb von 10sec am nächsten Sensor an so wird ein Timeout ausgelöst.

Ursachen:

- Schlitten hat sich verklemmt.

9. ID 8 kein freier Schlitten mehr vorhanden

Beschreibung: Kein freier Schlitten auf Nebenband 2 vorhanden

Ursachen:

- Kein Schlitten vorhanden

Folgen:

Bei allen Fehlern wird das Band angehalten. Es wird eine Meldung für den Mitarbeiter mit dem aufgetretenen Fehler am Bandsteuerungs-PC ausgegeben. Danach muss er den Fehler analysieren und beheben. Anschließend kann der Benutzer mit der Quittierungstaste K4 den Fehler quittieren und das Band kann wieder mit der Starttaste K2 gestartet werden.

6.4 Fehlerfälle für das Transportsystem

6.4.5 Bandsteuerungs-Fehler 5: Transportsystem kann nicht weiterarbeiten

Beschreibung:

Wenn der Operator im Fehlerzustand einen Retry auslöst, und dieser nicht erfolgen kann, wird ein Fehler 999 generiert. Ein Retry ist im Transportsystem nicht möglich, wenn das System neu hochgefahren wurde und aufgrund eines vorher unvollständig ausgeführten Befehls ein Recovery ausgeführt werden soll.

Folgen:

Fehlerbenachrichtigung an Steuerung. In den Nutzdaten ist der Befehl mit enthalten, der nicht erneut ausgeführt werden konnte.

6.4.6 Bandsteuerungs-Fehler 6: Verbindungsabbruch zur Steuerung

Bricht im Produktivbetrieb die Netzwerkverbindung zwischen Steuerung und Bandsteuerung zusammen, wird das gesamte System solange angehalten, bis ein zuständiger Operator das Problem behebt.

6.5 Fehlerfälle für die E/A-Station

6.5 Fehlerfälle für die E/A-Station

6.5.1 E/A-Station-Fehler 1: Befehl nicht verstanden

Beschreibung:

Ein von Steuerung gesendetes Kommando kann nicht identifiziert und damit nicht ausgeführt werden.

6.5.2 E/A-Station-Fehler 2: Befehl nicht ausgeführt

Beschreibung:

Ein von Steuerung gesendetes Kommando konnte nicht ausgeführt werden.

7 Wiederaufsetzen bzw. Wiederanfahren der Subsysteme (Recovery)

7.1 Allgemeiner Recovery-Vorgang

Unter Recovery ist das Wiederanlaufen eines Systems, z.B. der Lagersteuerung oder der Bandsteuerung nach einem ungewollten Stromausfall oder einem anderen unvorhergesehenen Ereignis und der erneute Versuch einer Auftragsausführung nach Behebung des Fehlers zu verstehen. Bei erneuter Versorgung mit Strom oder nach einer Fehlerbehebung in der Hardware soll das jeweilige System in einen betriebsbereiten Zustand fahren. Wurde ein Auftrag in seiner Bearbeitung unterbrochen, ist vom Operator zu entscheiden, ob der Auftrag fortgesetzt werden soll oder nicht. Falls der Operator den Auftrag nicht fortsetzen lässt, wird eine Fehlermeldung an die Steuerung geschickt, ansonsten werden die selben Nachrichten verschickt, die auch im Normalbetrieb verschickt würden.

Während der Abarbeitung eines Befehls können in den einzelnen Subsystemen Fehlerfälle auftreten. Es ist z.B. denkbar, dass während des Bestückungsvorgangs einer Produktpalette, oder während des Einlagervorgangs einer Lagerpalette die Stromversorgung für den Roboter, bzw. für das Lagersystem unterbrochen wird. Daraus ergibt sich die Problematik, dass zusammengehörige Vorgänge (z.B. physikalisches Einlagern einer Palette und Aktualisieren des Lagerbestands) unter Umständen nur teilweise ausgeführt wurden. Ziel des Recovery ist es, anhand von Protokollen diese Situation zu erkennen. Bei den Subsystemen wird das jeweilige Subsystem in eine Art Fehlerzustand versetzt und der Operator aufgefordert, Maßnahmen zu treffen, die das System wieder in einen fehlerfreien Zustand zurückversetzen. Bei Control wird beim Hochfahren des Systems überprüft, ob beim letzten Herunterfahren noch Befehle abzuarbeiten waren. Ist dies der Fall wird das System in den Recoverymodus versetzt. Der Operator wird aufgefordert zu entscheiden, ob der Auftrag weiter ausgeführt werden soll bzw. kann oder ob der Auftrag abgebrochen wird. Im letzteren Fall wird an JSAP eine Fehlermeldung zurückgegeben. Ansonsten wird anhand der Protokolldateien überprüft, welcher Befehl zum Schluss abgearbeitet wurde und es wird dort fortgefahrene, wo die Abarbeitung der Befehlswarteschlange unterbrochen wurde.

7.1.1 Schreiben der Log-Dateien

Jedes Subsystem verwendet zwei Log-Dateien. Eine temporäre Logdatei protokolliert nur den Nachrichtenaustausch zur Abarbeitung des aktuellen Befehls mit. Dazu wird als erstes der Befehl selbst, welcher von der zentralen Steuerung empfangen wurde, in die Datei geschrieben. Alle versendeten und empfangenen Nachrichten des Subsystems, die zur Abarbeitung des Befehls nötig sind, werden dann nach und nach an die Datei angehängt. Nachdem der gesamte Vorgang abgeschlossen, und die zentrale Steuerungseinheit darüber informiert wurde, werden

7.1 Allgemeiner Recovery-Vorgang

die mitprotokollierten Nachrichten der temporären Log-Datei in eine persistente Log-Datei geschrieben¹. Dort werden sie dauerhaft gespeichert. Der Inhalt der temporären Log-Datei wird schließlich gelöscht.

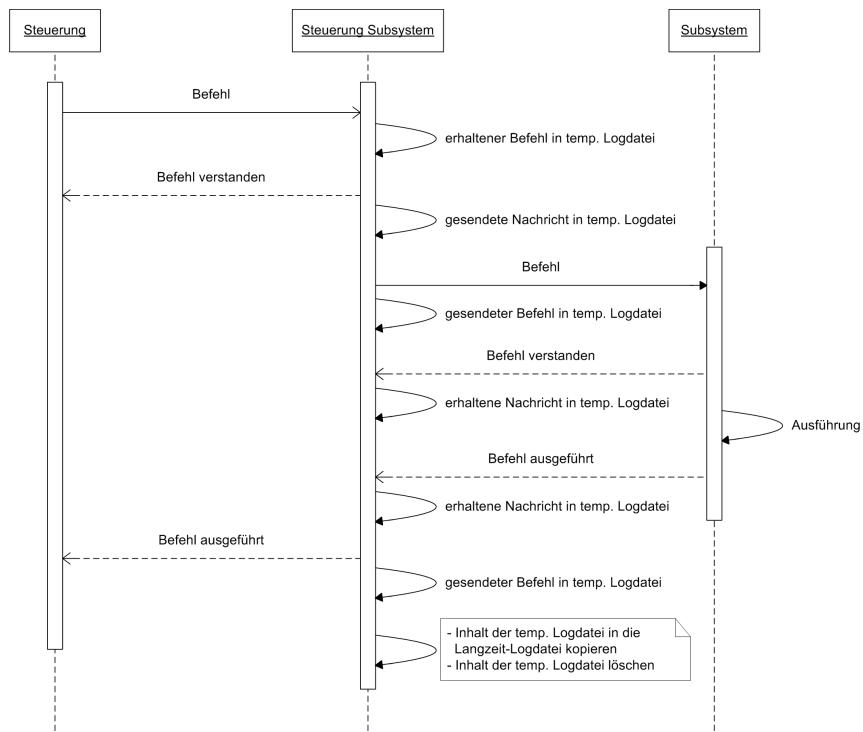


Abbildung 7.1: Schreiben der Log-Dateien

Die temporäre Logdatei für die Speicherung der Vorgänge eines Auftrags heißt `log_temp.txt`. Die persistente Datei zur Speicherung aller abgeschlossenen Aufträge heißt `log.txt`.

Aufbau `log_temp.txt`:

```

<Befehl von Steuerung an Subsystemsteuerung>
<Quittung an Steuerung: Nachricht verstanden(A001)>
<Elementarbefehl von Subsystemsteuerung an Subsystemhardware> // falls erforderlich
<Quittung von Subsystemhardware: Nachricht verstanden(A001)> // falls erforderlich
<Quittung von Subsystemhardware: Elementarbefehl ausgeführt(A002)> //falls erforderlich
<Quittung an Steuerung: Befehl ausgeführt(A002)>
  
```

Der inhaltliche Aufbau entspricht dadurch dem Aufbau der `log_temp.txt`-Datei.

¹Aktuell schreiben die meisten Teilsysteme alle Befehle bei Erhalt sofort sowohl in die temporäre als auch in die persistente Logdatei.

7.1 Allgemeiner Recovery-Vorgang

7.1.2 Systemstart im Recovery-Modus

Sollte ein Subsystem aufgrund eines Fehlerfalls einen Neustart durchführen und wurde der zuvor bearbeitete Vorgang nicht ordnungsgemäß beendet, wird dies bei der Initialisierung des Subsystems erkannt. Befinden sich zum Zeitpunkt der Initialisierung Einträge in der log_temp.txt-Datei, bedeutet dies, dass der zuletzt von der Steuerung erhaltene Befehl nicht korrekt ausgeführt wurde, da die log_temp.txt-Datei nach einem korrekt abgeschlossenen Vorgang geleert worden wäre.

Abbildung ?? zeigt die möglichen Fehlerfälle:

- 1: Das übergeordnete System stürzt ab, nachdem es einen Befehl an das untergeordnete System geschickt hat.
 - 2: Das untergeordnete System stürzt während der Bearbeitung eines Befehls ab.
- *: Eines der Systeme stürzt während der Übertragung einer Nachricht bzw. beim Eintrag der Nachricht in die log_temp.txt-Datei ab.

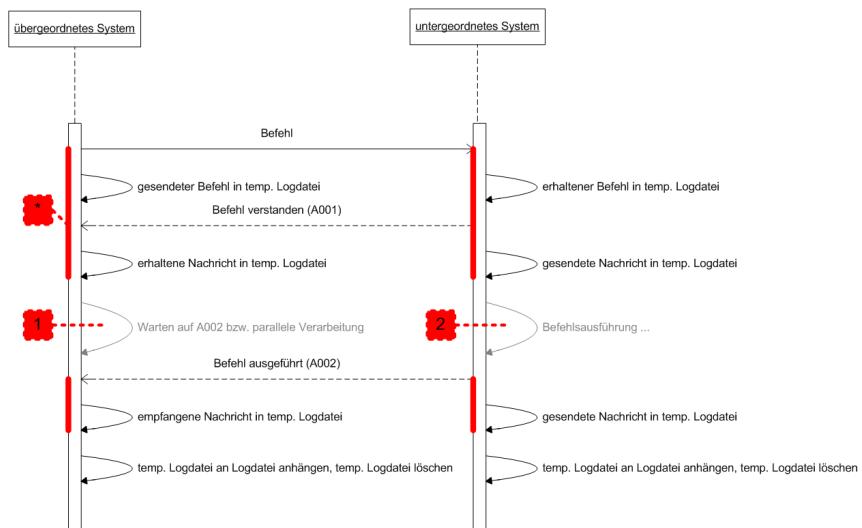


Abbildung 7.2: Mögliche Fehlerfälle, die zum Recovery führen

7.1 Allgemeiner Recovery-Vorgang

Durch Analyse der Datei kann das System entscheiden, wo der Fehler aufgetreten ist:

- 1: Im Fehlerfall 1 wartet das übergeordnete System auf die fehlenden Nachrichten.
- 2: Im Fehlerfall 2 muss der Operator am untergeordneten System entscheiden, ob der in der log_temp.txt-Datei gespeicherte Befehl erneut ausgeführt werden soll, oder nicht.

Soll der Befehl nicht erneut ausgeführt werden, erhält das übergeordnete System eine entsprechende Fehlermeldung (aktuell F999).

Beschließt der Operator hingegen, den Befehl erneut auszuführen, kann der Vorgang entweder gelingen oder scheitern. Das übergeordnete System erhält anschließend eine positive bzw. negative Quittung.

- *: Wenn ein Fehler während der Netzwerkübertragung bzw. beim Eintrag der Nachricht in die log_temp.txt-Datei aufgetreten ist, ist das Handling deutlich komplexer:

1. Der Übertragungsfehler wird von der Netzwerkschicht erkannt.

Das sendende System wartet den Wiederaufbau der Verbindung ab und schickt die Nachricht ab.²

2. Der Übertragungsfehler wird von der Netzwerkschicht nicht erkannt bzw. der Fehler tritt erst beim Eintrag in die log_temp.txt-Datei auf.

In diesem Fall ist es denkbar, am untergeordneten System die Nachrichten A001 und A002 nach Rückfrage beim Operator erneut senden zu lassen, wenn wenigstens der Befehl empfangen wurde. Wenn der Befehl nicht übertragen wurde, kann das übergeordnete System den Befehl nach einem Timeout erneut senden – ebenfalls nach Rückfrage beim Operator.^{3 4}

²Hinweis: Das steht im Widerspruch zur ursprünglichen Spezifikation

„Sollte beim Versenden einer Quittung an die Steuerung aufgrund eines Verbindungsabbruchs ein Timeout auftreten, dann wird dem Operator durch eine entsprechende Fehlermeldung mitgeteilt, dass das Subsystem nach Bestätigung dieser Meldung kontrolliert herunterfahren wird.“

ist aber wohl wie beschrieben implementiert worden.

³Hinweis: Diese Gedanken sind noch ungeordnet und nicht zu Ende gedacht!

⁴Anmerkung Plenk 21.6.06: Nach Diskussion mit Professor Scheidt haben sich folgende Punkte ergeben:

- Eine „Transaktion“ im Sinne von Bankenapplikationen zeichnet sich dadurch aus, dass sie, falls bei der (verteilten) Ausführung etwas schief geht, aus jedem Zustand automatisch vollständig rückgängig gemacht werden kann.
In der FDZ ist das nicht notwendig. Die einzelnen Kommandos können als atomar betrachtet werden. Evtl. ungültige Zustände nach Fehlern werden vom Operator manuell in gültige Zustände überführt.
 - Da die Kommunikation nicht 100% sicher dargestellt werden kann, ist es wohl einfacher von dem komplizierten Protokoll mit zwei Quittungen auf ein einfacheres an TCP/IP angelehntes Protokoll mit Timeout zurückzugehen:
 - a) Kommandos werden – wie hier spezifiziert – mit Zeitstempel und Nummer eindeutig gekennzeichnet.
 - b) Im fehlerfreien Fall wird ein Kommando nach vollständiger Ausführung mit einer Quittung bestätigt.
 - c) Wenn eines der beiden Systeme abstürzt, wird es entsprechend dem spezifizierten Recoveryverhalten neu gestartet.
 - d) Wenn eine Nachricht nicht zugestellt werden kann und das vom entsprechenden System bemerkt wird, wiederholt das System den Sendevorschlag nach Wiederherstellen der Netzwerkverbindung.
 - e) Wenn eine Nachricht nicht zugestellt werden kann und das vom entsprechenden System nicht bemerkt wird, sendet es die Nachricht nach einem festgelegten Timeout – der länger als die längste Ausführungszeit eines Kommandos sein sollte – erneut, aber mit gleichem Zeitstempel und gleicher Nummer.
- Wird eine Nachricht mit gleichem Zeitstempel und gleicher Nummer mehrfach empfangen, werden die

7.1 Allgemeiner Recovery-Vorgang

Abschließend werden die Informationen der log_temp.txt-Datei in die log.txt-Datei übernommen und die log_temp.txt-Datei geleert. Das Subsystem befindet sich nun wieder in einem betriebsbereiten Zustand. Die zentrale Steuereinheit muss über die weitere Vorgehensweise entscheiden.

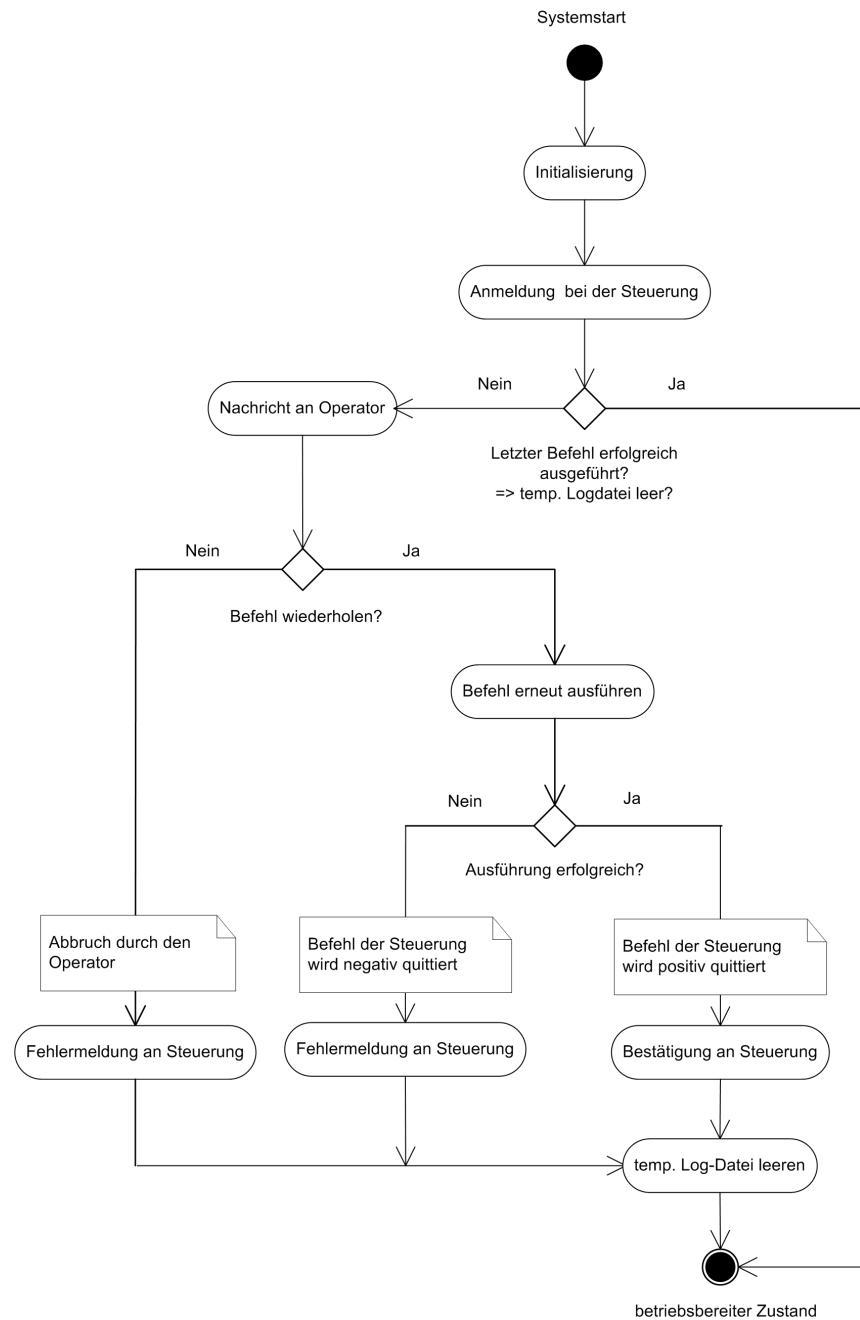


Abbildung 7.3: Systemstart im Recovery-Modus

Wiederholungen ignoriert.

7.2 Recoverybearbeitung der Steuerung

Für die Steuerung existiert eine Log-Datei für die versendeten und empfangenen Nachrichten und eine, oder mehrere Listen mit den noch zu versendenden Befehlen. Um Verwechslungen auszuschließen wird im folgenden die Datei mit den bereits versendeten Nachrichten als Logdatei und die mit den noch zu versendenden Befehlen als Queue bezeichnet.

Für die Steuerung wird nicht der unter ?? beschriebene, Ansatz für das Loggen verwendet, sondern es werden alle Befehle, die für das Abarbeiten eines Auftrages nötig sind, frühestmöglich in die Queue geschrieben. Sobald ein Befehl ausgeführt wurde, wird er in die persistente Logdatei geschrieben. Da die Lagersteuerung die angeforderte Anzahl von Smarties unter Umständen nicht auf einmal auslagert, muss die Queue ggf. während der Abarbeitung eines Auftrages verändert werden.

7.2.1 Log-Datei für Kommunikation JSAP - Steuerung (Log_Js2st)

Der Nachrichtenaustausch zwischen JSAP und der Steuerung kann im Idealfall wie folgt aussehen:

```
< JSAP schickt Auftrag an Steuerung >
< Steuerung schickt an JSAP "Auftrag verstanden" >
< Steuerung schickt Quittung an JSAP "Auftrag wird produziert" >
< Steuerung schickt Quittung an JSAP "Auftrag ausgeführt" >
```

7.2 Recoverybearbeitung der Steuerung

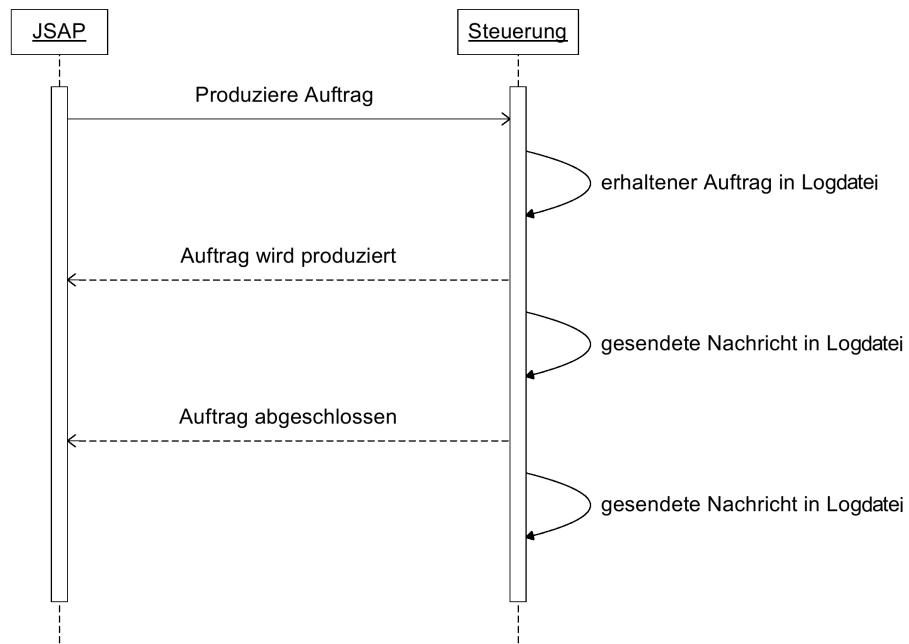


Abbildung 7.4: Schreiben der Log-Datei für die Kommunikation zwischen JSAP und der Steuerung

Die Grafik ?? veranschaulicht nochmals den Vorgang des Schreibens einer Log-Datei für die Kommunikation zwischen JSAP und der Steuerung.

Nähere Erläuterungen zu den Nachrichten, die von der Log-Datei geschrieben werden, sind unter dem Abschnitt ?? nachzulesen.

Mit diesem Protokoll kann der Zustand des Auftrags im Allgemeinen erläutert werden.

7.2 Recoverybearbeitung der Steuerung

7.2.2 Queue für noch abzuarbeitenden Befehle

Da der Ablauf der Steuerung komplizierter als der eines Subsystems ist, wird das Logging nicht wie in der Spezifikation beschrieben realisiert. Statt der temporären Log-Datei wird es eine Queue(Warteschlange) geben, in der alle Befehle, wie auch Unteraufträge abgespeichert werden. Die empfangenen Nachrichten, wie zum Beispiel A001 und A002, werden gleich in die persistente Log-Datei geschrieben, da sie als erfolgreich abgearbeitet gelten. Wenn ein Befehl vollständig ausgeführt wurde, wird der nächststehende Befehl aus der Warteschlange bearbeitet. Wenn die Queue keine weiteren Befehle enthält, bekommt JSAP eine Rückmeldung, dass der Auftrag ausgeführt wurde.

Inhalt der Queue:

```
<Befehl an Bandsteuerung>
<Befehl an Lagesteuerung>
<Befehl an Bandsteuerung>
<Befehl an Robotersteuerung>
.
.
.
<Befehl an Bandsteuerung>
<Befehl an EAstation>
```

7.2 Recoverybearbeitung der Steuerung

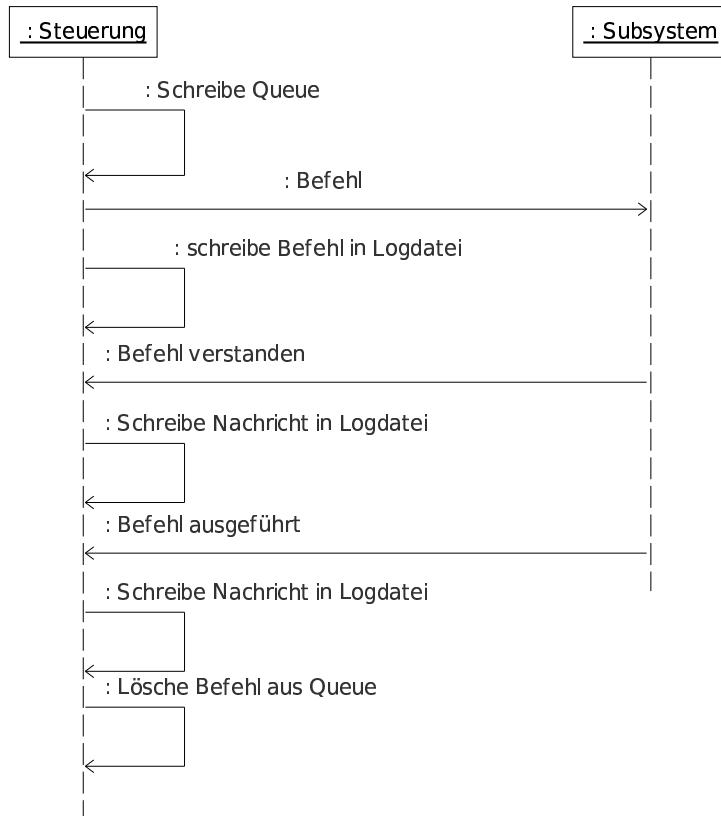


Abbildung 7.5: Schreiben der Queue–Datei am Beispiel der Kommunikation zwischen Steuerung und einem Subsystem

Die Grafik ?? veranschaulicht nochmals den Vorgang des Schreibens von Queue- und Log-Datei für die Kommunikation zwischen der Steuerung und einem Subsystem. Dabei wird für jedes Subsystem die selbe Log-Datei verwendet. Somit gibt es nur eine Log-Datei für die Kommunikation zwischen der Steuerung und den Subsystemen.

Wird ein Befehl vom Subsystem abgearbeitet, erhält die Steuerung eine dazugehörige Quittierung. Wird vom Subsystem die Quittierung "Befehl ausgeführt" an die Steuerung geschickt, so wird der nächste Befehl aus der Queue ausgeführt und in die Log-Datei geschrieben.

7.2.3 Aufbau der Queue datei

An erster Stelle befindet sich immer die ID des dazugehörigen Jobs. Diese kann entweder Restore, Select, Receive, Send, Simulator oder der Anfang eines Befehls sein (siehe ??), also der Teil vor der MessageID.

Anschließend folgt immer das Trennzeichen #.

Wenn der Job auch einen Parameter in Form einer Map hat, folgt dieser in der Form

`<key>=<value><|>`

7.2 Recoverybearbeitung der Steuerung

<key> und <value> sind Nonterminalsymbole. Die verwendeten Keys sind in der Datei `fdz-job.h` definiert. Das Trennzeichen <|> folgt auch, wenn anschließend kein neues Wertepaar mehr folgt.

Um überprüfen zu können, ob die Datei vollständig gespeichert wurde wird das End of File Flag `#~EOF~#` angehängt

Der Inhalt einer Queue datei könnte beispielsweise wie folgt aussehen:

```
STSIK002#currentColor=g<|>currentQuantity=000<|>
STSIK002#currentColor=b<|>currentQuantity=000<|>
STSIK002#currentColor=w<|>currentQuantity=000<|>
STSIK002#currentColor=y<|>currentQuantity=000<|>
#~EOF~#
```

7.2.4 Vorgang für das Recovery

Beim Start der Steuerung wird zuerst überprüft, ob in der Queue noch ein Befehl zum abarbeiten vorhanden ist. Ist dies nicht der Fall, so kann der Betrieb normal fortgesetzt werden. Befindet sich jedoch mindestens ein Befehl in Queue, so muss überprüft werden, ob dieser bereits versendet wurde. Dies ist der Fall, wenn er auch in der Logdatei steht. Wurde zu dem Befehl kein A001 empfangen, so muss er erneut gesendet werden, ansonsten wird auf das A002 gewartet. Da im Recoverymodus keine neuen Befehle versendet werden können, muss die Bestandsdatei vorhanden sein, ansonsten beendet sich die Steuerung und teilt dem Operator den Fehler mit.

Beispiel: Sollte nach dem Befehl die Nachricht "Quittierung Befehl verstanden" in der Log-Datei stehen, wartet die Steuerung (nach dem Wiederanlauf) darauf, dass das Subsystem diesen Befehl abarbeitet und eine Quittierung schickt.

Sollte kein Befehl beim Recovery in der Queue-Datei stehen, wird die Steuerung den nächsten Befehl an ein Subsystem hinausschicken.

7.2 Recoverybearbeitung der Steuerung

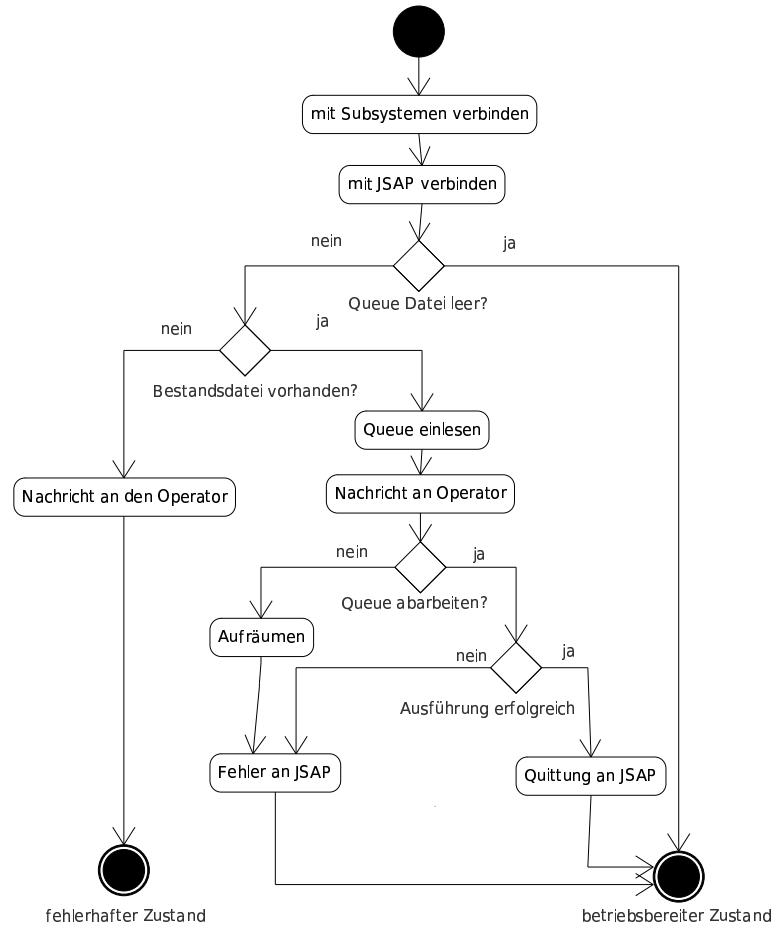


Abbildung 7.6: Ablauf beim Start von Control

7.3 Recoverybearbeitung des Robotersystems

Nach einem Absturz, Stromausfall oder Notaus muss das System wieder in einen möglichst funktionsfähigen Zustand hochfahren können. Hierzu werden wir Log-Dateien nutzen. Die Dateien werden im Verzeichnis des jeweiligen Steuerprogrammes (Sr oder ro) gespeichert.

7.3.1 Änderungen entgegen dem allgemeinen Recovery-Verhalten in ??

Für die Robotersteuerung werden die Nachrichten, die mit dem Roboter und mit der Steuerung ausgetauscht werden, in getrennten Logfiles gespeichert.

Im Recovery-Fall wird dann aus dem log-File für die Nachrichten von der Steuerung das letzte Kommando extrahiert, das nicht mit A002 oder einem Fehler quittiert wurde.

Diese Kommando wird genau so verarbeitet, als wenn es gerade von der Steuerung erhalten worden wäre.

Die „Intelligenz“ für das Recovery liegt in der Routine, welche die Nachrichten an den Roboter sendet:

Zunächst wird geprüft, ob das Kommando bereits im Logfile vorliegt. Wenn nicht, wird es geloggt und wie im normalen Ablauf ausgeführt.

Wenn das Kommando bereits vorliegt, wird geprüft, ob bereits ein Ack1 vorliegt. Wenn nicht, wird das Kommando wie im normalen Ablauf ausgeführt.

Wenn A001 bereits vorliegt, wird geprüft, ob auch A002 vorliegt. Wenn es bereits vorliegt, kehrt die Sendemethode ohne Aktion zurück (der Befehl wurde offensichtlich bereits Fehlerfrei ausgeführt).

Wenn A002 nicht vorliegt, wartet die Methode auf A002 oder eine Fehlermeldung von ro.

ACHTUNG !!! Das Recovery geht davon aus, dass nur der übergebene Stand der Paletten gespeichert wird. Da die Intelligenz bei der Sendemethode liegt, darf eine Veränderung der Paletten während der Bearbeitung NICHT gespeichert werden, sonst scheitert das Recovery!!!⁵

7.3.2 Benennungen der Recovery-Files

Es werden folgende Namen für die Dateien vorgesehen:

- **log_SR.txt**
für die Nachrichten die zwischen Steuerung und Robotersteuerung ausgetauscht wurden
- **log_ro.txt**
für die Nachrichten die zwischen Robotersteuerung und Roboter ausgetauscht wurden

⁵Das ist aktuell nicht der Fall! Während Bestückung wird jede einzelne Smartiebewegung in Bestandsdateien geschrieben. Sinn der Warnung ist nicht klar. Speicherung der einzelnen Bewegung nötig, um Fehlerfälle wie doppelte Bestückung einer Position zu vermeiden. Lediglich die bereits geänderte LP-Matrix darf im Recovery-Fall (Absturz während Bestückung) nicht ausgewertet werden (da sonst falsche Elementarbefehle generiert würden). Stattdessen wird die Matrix vor Bestückungsbeginn herangezogen, um alle Elementarbefehle erneut zu erzeugen. Da die erfolgreich abgearbeiteten Elementarbefehl noch in log.ro.txt stehen, werden sie übersprungen.

7.3 Recoverybearbeitung des Robotersystems

- **log.txt**
für die persistente Speicherung aller Nachrichten (Sr & ro)
- **bestand_res.txt**
für die Daten der eingelagerten Lagerpaletten (nur Sr, ro hat und benötigt diese Daten nicht)
- **bestand_X.txt**
für die Daten der eingelagerten Produktpalette (nur Sr, ro hat und benötigt diese Daten nicht)
- **bestand_assembly.txt**
für die Daten der eingelagerten Lagerpalette vor Beginn einer Bestückung (nur Sr, ro hat und benötigt diese Daten nicht)⁶

Zudem bedeutet eine leere oder nicht vorhandene Datei, dass keine Paletten eingelagert sind oder die letzte Befehlskette erfolgreich abgeschlossen wurde.

Sollte also ein kompletter Reset des Systems durchgeführt werden, genügt es diese Dateien zu löschen. Siehe "Allgemeiner Recovery-Vorgang" ??.

Grundlegend sind zwei Informationen zu speichern:

7.3.3 ID-Nummern und Matrixinformationen der eingelagerten Lagerpalette.

Zusätzlich muss der Arbeitsplatz der eingelagerten LP gespeichert werden. Damit kann jederzeit mit vorhandenen Paletten weitergearbeitet werden.

Es wird 1 Zeile in der Datei bestand_res.txt in folgendem Format gespeichert:

{AB} {rgybw-} * {91} ;

{AB}

gibt Auskunft über den Lagerplatz der Palette auf dem Arbeitstisch

{rgybw-}*{91}

Palettenbelegung, 91 Zeichen für Belegung der LP.

Wenn die Datei bestand_res.txt vorhanden ist, bedeutet dies, dass eine Lagerpalette beim Roboter steht. Ist die Datei nicht vorhanden, steht keine Lagerpalette beim Roboter.

⁶Datei existiert nur während die Bestückung nicht vollständig abgeschlossen ist. Sie dient dazu, dass bei einem Recoveryfall, der während der Bestückung auftritt, die ursprüngliche Reihe der Elementarbefehle an den Roboter erneut erzeugt werden kann. bestand_res.txt kann hierzu nicht herangezogen werden, da durch die bereits entfernten Smarties, die vor dem Crash bestückt wurden, andere (ungültige) Befehle erzeugt werden würden.

7.3.4 Matrixinformation der eingelagerten Produktpalette.

Es wird 1 Zeile in der Datei bestand_X.txt in folgendem Format gespeichert, wenn eine Bestückung vorgenommen wird:

{X} {rgybw-}*{63};

{X}

gibt Auskunft über den Lagerplatz der Palette auf dem Arbeitstisch

{rgybw-}*{63}

Palettenbelegung, 63 Zeichen für Belegung der LP.

Wenn die Datei bestand_X.txt vorhanden ist, bedeutet dies, dass eine Produktpalette beim Roboter steht. Ist die Datei nicht vorhanden, steht keine Produktpalette beim Roboter. Ist die Datei leer, so hat noch keine Bestückung begonnen.

7.3.5 Erhaltene und mögliche Antworten.

Es werden die empfangen (Elementar)Befehle und gesendeten Nachrichten 1:1 in die Datei gespeichert; 1 Befehl/Antwort pro Zeile. Die Reihenfolge der Speicherung sollte der Reihenfolge der Kommunikation entsprechen: **Befehl -> Antwort [-> Antwort]**.

So kann zumindest die letzte Aktion des letzten Zustandes nachvollzogen werden. Es kann somit zu maximal 3 möglichen Fällen nach einem Neustart kommen:

- Die Datei ist leer, nicht vorhanden oder der letzte Eintrag in der Datei ist eine aller Antworten ausser A001
 - Neue Befehle können von der Steuerung empfangen werden, da der letzte Auftrag erfolgreich abgearbeitet wurde (A002 ist letzte Antwort), oder ein Fehler auftrat, was auch ein Ende des Auftrags bedeutet.
- Der zuletzt gespeicherte Eintrag in der Datei ist ein Befehl (K???)
 - Neuer Versuch, den gespeicherten Befehl auszuführen, ist möglich/erlaubt.
- Der zuletzt gespeicherte Eintrag in der Datei ist A001
 - Warten auf die Antwort des Roboters.⁷

Wird ein Befehl empfangen, so wird dieser sowohl in die entsprechende temporäre Logdatei (log_ro.txt / log_SR.txt) als auch in die persistente Logdatei (log.txt) eingetragen.

⁷Ursprünglicher Text: „Vermutlich ist ein Problem mit dem Roboter aufgetaucht, dies ist ein harter Fehler. Es darf nicht einfach der letzte Elementarbefehl ausgeführt werden und Fehler F999 muss an die Steuerung geschickt werden.“

Jedoch nicht richtig, da lt. Prof. Stöhr davon ausgegangen werden kann, dass die von ro gesendeten Antworten auch nach dem Neustart von Sr noch empfangen werden können. D.h. es gehen keine Nachrichten verloren, nur weil Sr abgestürzt ist.

7.4 Recoverybearbeitung des Lagersystems

Unter Recovery ist ausschließlich der Wiederanlauf des Lagers nach einem ungewollten Stromausfalls oder einem bereits quittierten Fehler zu verstehen. Bei erneuter Versorgung mit Strom oder nach Quittierung eines Hardwarefehlers soll das Lager in einen betriebsbereiten Zustand fahren. Wurde ein Auftrag in seiner Bearbeitung unterbrochen, ist vom Lageroperator zu entscheiden, ob der Auftrag wiederholt werden soll oder nicht.

In diesem Abschnitt wird beschrieben, wie das Lagersystem auf die bereits beschriebenen Fehlerfälle durch Initiierung des Recovery-Vorgangs reagiert. Es wird nicht nach der allgemeinen Recovery-Beschreibung verfahren, da bei der Analyse des übernommenen Softwarestands zur Steuerung der Lager-Hardware aufgefallen ist, dass dort das Recovery bereits implementiert ist. Bei der Besprechung aller nötigen Änderungen wurde beschlossen, diese Funktionen hardwarenah zu belassen. Die Abläufe zur Sicherung, Prüfung und zum Auslesen des Recovery-Files ist bereits in dieser Form implementiert, müssen aber an das neue Protokoll angepasst und in die neue Taskinteraktion integriert werden.

Das Lager führt nur ein eigenes Log in der Datei "RECOV.DAT". Aufgrund der begrenzten Verfügbarkeit von Speicher auf dem Lagerchip ist es nicht möglich, alle Vorgänge der Lagermechanik direkt durch den Steuerungsschip im Lager mitzuloggen. Es ist lediglich möglich Elementarbefehle mitzuloggen, die das Lager von der Lagersteuerung erhält. Die Elementarbefehle werden jeweils nur zwischen der positiven Quittierung des Erhalts (A001) bis zur positiven oder negativen Quittierung der Ausführung des Auftrags (A002, F00x) mitgeloggt.

Die Lagersteuerung führt ebenfalls ein Log und kann nach einem Absturz unter Beteiligung des Operators das abgebrochene Kommando fortsetzen. Da das unter ?? beschriebene Format der temporären Logdatei nur die gesendeten und empfangenen Nachrichten enthält kann damit nicht zuverlässig entschieden werden an welcher Stelle im Programm abgebrochen wurde. Deshalb verwendet die Lagersteuerung eine erweiterte temporäre Logdatei in der zusätzlich Statusinformationen als von 0 beginnende, fortlaufende Zahlenwerte abgespeichert werden. Um zu verdeutlichen dass diese Informationen optional sind wird // vorangestellt. Diese Informationen werden nicht in die persistente Logdatei geschrieben. (siehe ??)

Die folgende Beschreibung bezieht sich auf das Recover aufgrund eines Fehlers während einer aktiven Auftragsbearbeitung.

7.4 Recoverybearbeitung des Lagersystems

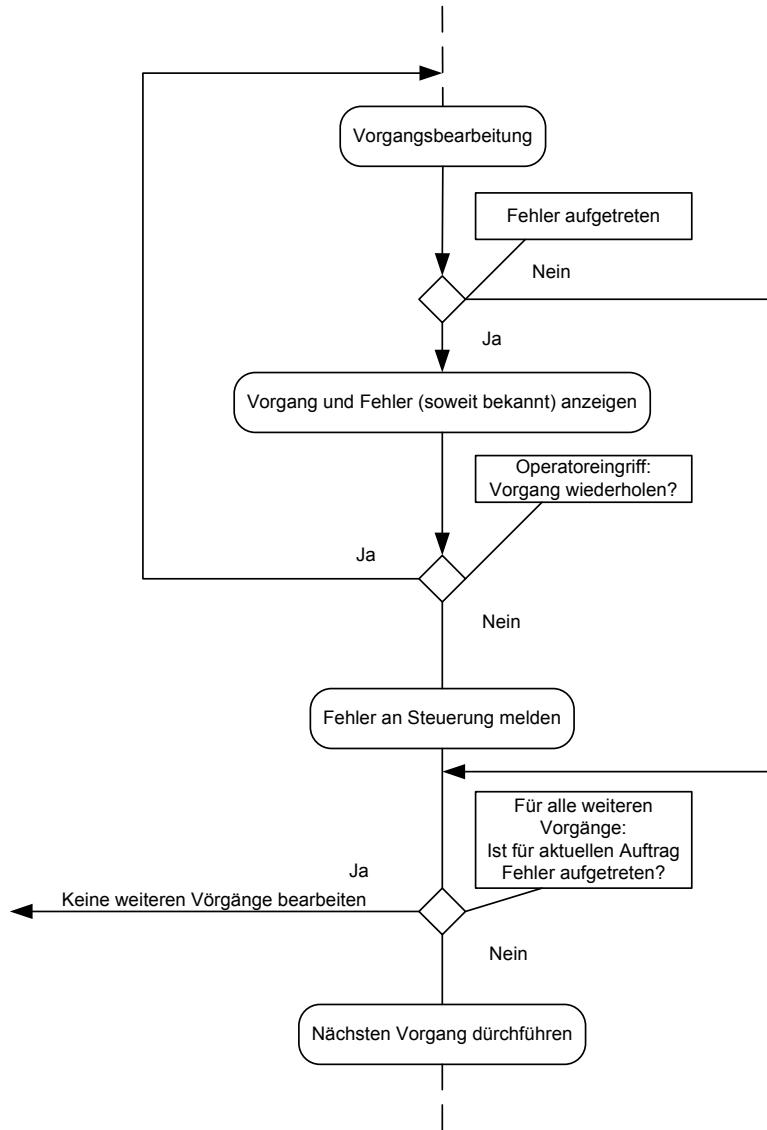


Abbildung 7.7: Fehlerbearbeitung während eines Auftrags

Der hier gezeigte Ablauf entspricht sowohl der Recoverybearbeitung im Lager als auch der Lagersteuerung. Grundsätzlich soll dem Lageroperator im Falle eines Fehlers während des Betriebs die Möglichkeit gegeben werden die Lagerhardware und den Zustand der Auftragsbearbeitung zu prüfen, ggf. Korrekturen an der Physik vorzunehmen, den Vorgang bei Lager und Lagersteuerung zu wiederholen oder fehlerhaft abzubrechen. Es ist aber nicht bei jedem Fehlerfall sinnvoll den Operator zu bemühen, da er manche Fehler nicht beheben kann. Zu diesen nicht durch den Operator behebbaren Fehler zählen die im vorangegangenen Kapitel als Allgemein beschriebenen Fehler:

- Fehler bei syntaktischem Aufbau des Elementarbefehls an das Lager

7.4 Recoverybearbeitung des Lagersystems

- Kommando/Befehl unbekannt
- Fehler bei lesendem und/oder schreibendem Zugriff auf den Lagerbestand in der Lagerbestandsdatei
- Fehler bei lesendem und/oder schreibendem Zugriff auf die zwischengespeicherten Befehle in den Log-Dateien (Lager und Lagersteuerung)

Weiterhin macht es bei folgenden Fehlern keinen Sinn, den Operator hinzuzuziehen, da es sich hier um programmier- bzw. ablauspezifische Fehler handelt. Diese können nicht durch den Operator beeinflusst werden und ergeben auch bei mehrfacher Wiederholung des Vorgangs das gleiche Ergebnis. Die Beschreibung der Ursache des jeweiligen Fehlers ist in der Fehlerfallbeschreibung vor diesem Kapitel erläutert.

- kein Palettenstellplatz frei bei Anfrage zur Einlagerung einer Palette
- leere Lagerpalette zur Einlagerung übergeben
- kein Bestand für Auslagerungsauftrag
- Auftrag zur Einlagerung einer Produktpalette wurde übertragen
- Fehler Auftrag an das Lager zur Auslagerung eines leeren Fachs und Einlagerung ein belegtes Fach geschickt

Letztlich macht es nur im Falle einer mechanischen Störung und bei Erkennung eines Recovery-Falls Sinn, den Operator hinzuzuziehen. Bei einer Störung während eines Ein- bzw. Auslagerauftrags gibt das Lager den Fehler auf dem LCD aus. Der Operator kann mit dieser Information und einer Untersuchung des aktuellen Lagerzustands nachvollziehen, wo der Fehler liegt. Wenn es sich um einen kleinen Fehler handelt (z.B. verklemmte Palette, Smartie blockiert Sensor,...) kann der Operator das Lager (nur das Lager!) stromlos schalten und den Fehler beheben. In dieser Meldung wird dem Operator zur Wahl gestellt, den Fehler positiv zu quittieren oder den Auftrag abzubrechen. Wird der Fehler negativ quittiert fährt das Lager sofort herunter, um weitere Schäden an der Hardware zu vermeiden. Eine Quittung wird nicht an die Lagersteuerung geschickt und der Auftrag bleibt in der Datei "RECOV.DAT" erhalten. Im Falle der positiven Quittierung wird versucht, das Lager in Grundstellung zu fahren. Anschließend tritt der Recovery-Fall genau so ein, wie beim Hochfahren des Lagers. D.h. es wird erkannt, daß sich ein Elementarbefehl in der Log-Datei befindet. Dies wird dem Operator über das LCD angezeigt. Er kann nun über die Handsteuerung wählen, ob der Auftrag wiederholt, oder mit einer Fehlerquittung an die Lagersteuerung abgebrochen werden soll. Die Lagersteuerung muss nach wie vor auf die Quittierung des Auftrags warten. Sollte auch Sie zwischenzeitlich ausgefallen sein, muss diese wieder in den Zustand zum Warten auf eine Vollzugsquittung gefahren werden.

7.5 Recoverybearbeitung des Transportsystems

7.5 Recoverybearbeitung des Transportsystems

Ablauf siehe EArecovery.

7.6 Recoverybearbeitung der E/A-Station

Der Recoveryvorgang läuft prinzipiell genauso ab wie bei der Steuerung. Hinzu kommt lediglich, dass die Bestandsdatei im Gegensatz zu der Steuerung aus dem Recovery heraus wieder hergestellt werden kann.

Teil III

Simulationsumgebung

8 Konstruktion der Simulation

Die Simulationsumgebung ist eine Java-Anwendung, die die Tätigkeiten aller Teile in der Fabrik der Zukunft kennen und gezielt simulieren soll.

Sie gliedert sich in die beiden Unterprogramme

- Funktionssimulator, der die untergeordneten FDZ-Teile simuliert und auf Befehle antwortet und
- Kommandogenerator, der die Befehle der übergeordneten FDZ-Teile generiert.

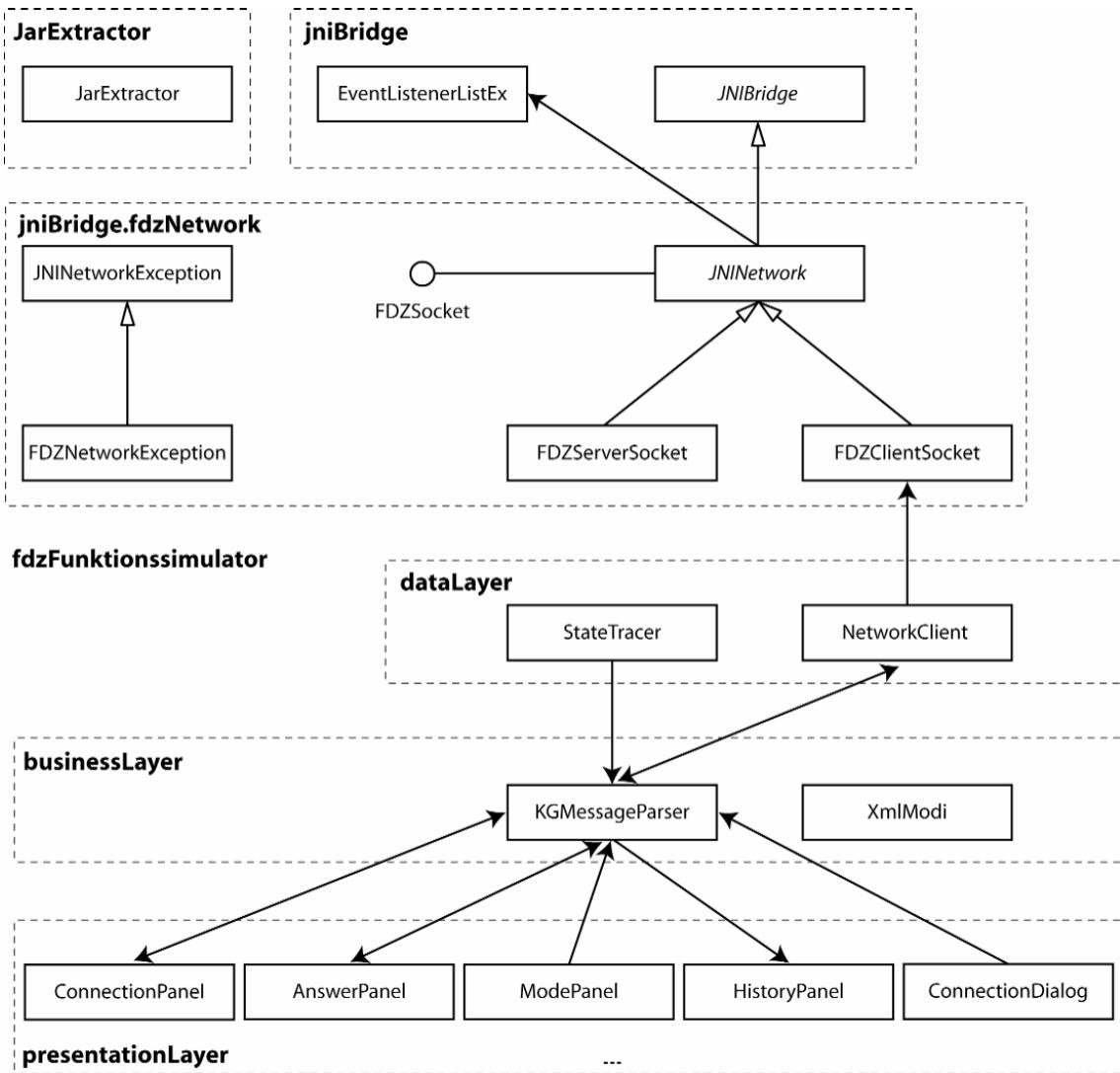
Vom Aufbau gliedern sich der Kommandogenerator und der Funktionssimulator jeweils in drei Packages:

- presentationLayer, beinhaltet die Klassen für die Oberflächen und ihre Module
- businessLayer, bereitet die jeweils empfangenen Daten auf und gibt sie an den anderen Layer weiter
- dataLayer, ist für den Datenaustausch mit der jniBridge zuständig

8.1 Funktionssimulator

Der Funktionssimulator simuliert die untergeordneten FDZ-Teile, wie z.B. die Steuerung, die Lagersteuerung, Robotsteuerung, Mit seiner Hilfe können Anfragen von diesen Teilen gezielt beobachtet und beantwortet werden.

8.1 Funktionssimulator



Die Übersicht zeigt den Aufbau des Funktionssimulators. So kann man die Verbindungen der einzelnen Klassen packageübergreifend entnehmen. Aus dem Package „presentationLayer“ sind nur die Klassen aufgelistet, die mit Klassen aus dem Package „businessLayer“ in Verbindung stehen. Des Weiteren werden auch nicht die Verbindungen der Klassen im Package „presentationLayer“ dargestellt. Der komplette Aufbau von „presentationLayer“ ist im entsprechenden Klassendiagramm weiter unten zu finden. Die eigentliche Verbindung zwischen Kommandogenerator und der „jniBridge“ wird über die Klasse „NetworkServer“ hergestellt.

8.1.1 Package - presentationLayer

Mit dem Package `presentationLayer`, werden die Oberflächen, für die FDZ-Teile im Funktionssimulator dargestellt. Er dient der Kommunikation zwischen dem System und dem Anwender, damit entspricht er einem HMI (Human Machine Interface).

8.1 Funktionssimulator

8.1.1.1 AnswerPanel

Bei dem AnswerPanel handelt es sich, um das Gegenstück zum CommandPanel des Kommandogenerators.

Je nach Einstellung des Funktionssimulators, arbeitet das AnswerPanel unterschiedlich. Wurde der automatische Modus gewählt, werden Antworten selbstständig gegeben (per Default-Einstellung immer positive Antwort). Wurde der manuelle Betriebsmodus gewählt, werden alle möglichen Antworten angezeigt. Wird daraus eine Antwort zum Senden ausgewählt, so wird wenn nötig vor dem Senden der „InputDialog“ aufgerufen, um die Rückgabeparameter festzulegen. Danach kann die Antwort gesendet werden.

8.1 Funktionssimulator

C AnswerPanel

- ❑ but_change: JButton
- ❑ but_send: JButton
- ❑ commands: String
- ❑ data: XmlResponse
- ❑ descriptions: String
- ❑ historyPanel: HistoryPanel
- ❑ input: Vector
- ❑ inputDialog: InputDialog
- ❑ isModeAutomatic: boolean
- ❑ jsapiInputDialog: JSAPIInputDialog
- ❑ list_answers: JList
- ❑ madeCommand: String
- ❑ madeDescription: String
- ❑ manipulationPanel: ManipulationPanel
- ❑ messageParser: FSMessageParser
- ❑ messageType: String
- ❑ modePanel: ModePanel
- ❑ pm_commandoAuswahl: JPopupMenu
- ❑ receivedMessage: String
- ❑ roboTerSmartieLayoutDialog: RoboterSmartieLayoutDialog
- ❑ sendError: boolean
- ❑ serialVersionUID: long
- ❑ showCommands: boolean
- ❑ smartieLayoutDialog: SmartieLayoutDialog
- ❑ useManipulation: boolean

C AnswerPanel(in historyPanel: HistoryPanel, in manipulationPanel: ManipulationPanel)

- abortPressed()
- buildDescriptionVector()
- buildInputVector()
- buildMessageVector()
- createCommand()
- getButtonPanel(): JPanel
- getMadeCommand(): String
- getReceivedMessage(): String
- init()
- initButtons()
- initJList()
- removeData()
- sendCommand()
- setAnswersToSend()
- setButtons()
- setCommands(in commands: Vector)
- ▲ setData
- setMadeCommand(in command: String, in description: String)
- setModePanel(in modePanel: ModePanel)
- setParser(in messageParser: FSMessageParser)
- showCommands()
- showDescriptions()
- showInputParameter(): boolean
- useManipulation(in value: boolean)

8.1 Funktionssimulator

8.1.1.2 Applicationconstants

Ein Interface, das Konstanten (z. B. Pfade, Default-Ports) beinhaltet auf welche die verschiedenen Klassen zurückgreifen können.

«interface»	
Applicationconstants	
`f	CMD_ABORT: String
`f	CMD_CONNECT: String
`f	CMD_CONTROL: String
`f	CMD_DISCONNECT: String
`f	CMD_IO: String
`f	CMD_JOCONTROL: String
`f	CMD_JSAP: String
`f	CMD_NEXT: String
`f	CMD_NO: String
`f	CMD_NUMBERS: String
`f	CMD_OK: String
`f	CMD_RESET: String
`f	CMD_RGYBM: String
`f	CMD_RGYBW: String
`f	CMD_RGYBW_X: String
`f	CMD_ROBOT: String
`f	CMD_ROBOTCONTROL: String
`f	CMD_SLEIGHPOS: String
`f	CMD_STORAGE: String
`f	CMD_STORAGECONTROL: String
`f	CMD_TRANSPORT: String
`f	CMD_TRANSPORTCONTROL: String
`f	CMD_YES: String
`f	DEBUG: String
`f	DEBUG_PATH: String
`f	DEFAULT_IP: String
`f	DEFAULT_PORT: String
`f	DEFAULT_PORT_CO: String
`f	DEFAULT_PORT_IO: String
`f	DEFAULT_PORT_JOCO: String
`f	DEFAULT_PORT_RO: String
`f	DEFAULT_PORT_ROCO: String
`f	DEFAULT_PORT_ST: String
`f	DEFAULT_PORT_STCO: String
`f	DEFAULT_PORT_TR: String
`f	DEFAULT_PORT_TRCO: String
`f	defaultToolkit: Toolkit
`f	errorDialog: ErrorDialog
`f	FILE_CONNECTION: String
`f	FILE_IO_XML: String
`f	FILE_IO_XML_EXTERN: String
`f	FILE_JSAP_XML: String
`f	FILE_JSAP_XML_EXTERN: String
`f	FILE_MESSAGE_STREAM: String
`f	FILE_ROBOT_XML: String
`f	FILE_ROBOT_XML_EXTERN: String
`f	FILE_STORAGE_XML: String
`f	FILE_STORAGE_XML_EXTERN: String
`f	FILE_TRANSPORT_XML: String
`f	FILE_TRANSPORT_XML_EXTERN: String
`f	ICON_CONTROL: String
`f	ICON_JSON: String
`f	ICON_INFO: String
`f	ICON_IO: String
`f	ICON_RIGHT: String
`f	ICON_ROBOT: String
`f	ICON_STORAGE: String
`f	ICON_TRANSPORT: String
`f	JAR_NAME: String
`f	jarExtractor: JarExtractor
`f	OUT_ABORT: String
`f	OUT_CONNECTED: String
`f	OUT_DATA_CHANGED: String
`f	OUT_ERROR: String
`f	OUT_ERROR_UNKNOWN: String
`f	OUT_LOST: String
`f	OUT_MANUALLY_DISCONNECTED: String
`f	OUT_READY: String
`f	OUT_RECEIVED: String
`f	OUT_SENT: String
`f	OUT_TRY: String
`f	OUT_WAITING: String
`f	OUT_WAITING_CON: String
`f	XML_TYPE_ERROR: String
`f	XML_TYPE_SUCCESS: String

8.1 Funktionssimulator

8.1.1.3 ConnectionDialog

Dieser Dialog erscheint wenn beim Start oder Ändern der Verbindungsoptionen (Ip-Adresse, Port) versucht wird, eine Verbindung zum entsprechenden FDZ-Teil aufzubauen. Es ist jederzeit möglich den Verbindungsauftbau manuell abzubrechen.



8.1.1.4 ConnectionOptionsDialog

Es werden die momentanen Verbindungseinstellungen (IP-Adresse, Port) dargestellt, zu denen versucht wird eine Verbindung aufzubauen bzw. eine aufgebaut ist. Ist eine Verbindung zustande gekommen, lassen sich die Verbindungseinstellungen hier ändern.

Die getätigten Einstellungen werden in einer Datei gespeichert und beim nächsten Start wieder zum Herstellen einer Verbindung genommen. Ist beim Versuch die Daten zu laden keine Datei vorhanden oder die Einstellungen noch nicht darin gespeichert, wird als IP-Adresse 127.0.0.1 und als Port der in der Spezifikation festgelegte Port hergenommen.

Die Syntax der IP-Adresse wird im „nextButtonPressedHandler“ untersucht.

G ConnectionOptionsDialog

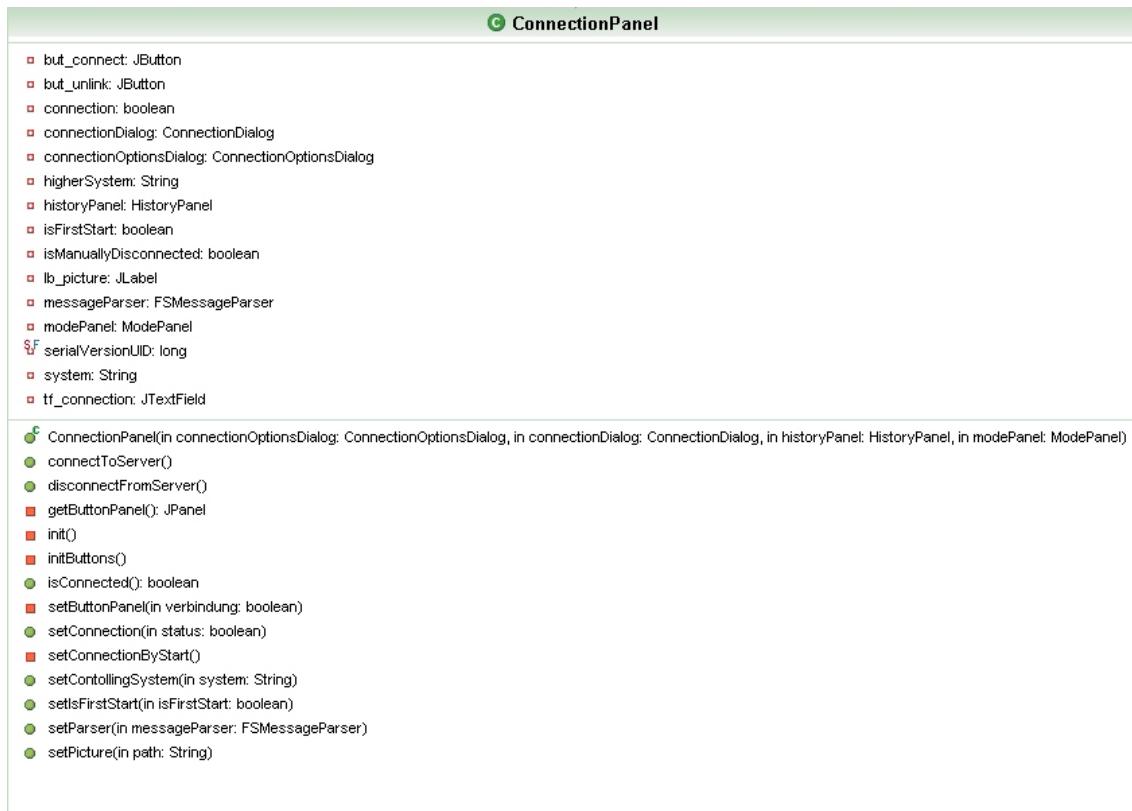
- △ but_abort: JButton
- △ but_ok: JButton
- △ connectionDialog: ConnectionDialog
- ▣ connection_properties: Properties
- §F serialVersionUID: long
- △ simulator: String
- △ st_ip: String
- △ st_port: String
- △ st_tmplIP: String
- △ st_tmppPort: String
- △ tf_ip: JTextField
- △ tf_port: JTextField

- C ConnectionOptionsDialog(in owner: MainFrame, in model: boolean)
- getIP(): String
- getPort(): String
- ▣ init()
- ▣ initButtons()
- loadConnectionProperties()
- ▣ positionDialog()
- saveConnectionProperties()
- setConnectionDialog(in connectionDialog: ConnectionDialog)
- setPort(in simulator: String)
- showDialog(in simulator: String)

8.1 Funktionssimulator

8.1.1.5 ConnectionPanel

Zeigt den momentanen Verbindungsstatus (verbunden / nicht verbunden) an. Zusätzlich wird der Name des aktuellen Funktionssimulators und der des FDZ-Teils, mit dem kommuniziert werden soll angezeigt. Außerdem besteht die Möglichkeit die Verbindung zu trennen bzw. wieder aufzubauen.



8.1.1.6 ErrorDialog

Erstellt für aufgetretene Fehler den richtigen Dialog. D.h. kreiert einen Dialog mit dem jeweiligen Fehlertext und der daraus entstehenden Möglichkeit darauf zu reagieren.

8.1 Funktionssimulator

C ErrorDialog

- ❑ but_no: JButton
- ❑ but_ok: JButton
- ❑ but_yes: JButton
- ❑ content: JPanel
- ❑ currentDialog: int
- ❑ lb_error: JLabel
- ❑ serialVersionUID: long
- ❑ sp_error: JScrollPane
- ❑ tp_error: JTextPane

- **ErrorDialog(in owner: JFrame, in modal: boolean)**
- actionPerformed(in e: ActionEvent)
- ❑ getButtonPanel(): JPanel
- ❑ init()
- ❑ positionDialog()
- showAlertDialog(in title: String, in errorMessage: String)
- showAlertDialog(in errorMessage: String)
- showFatalAlertDialog(in errorMessage: String)
- showFatalAlertDialog(in title: String, in errorMessage: String)

8.1.1.7 FSChoiceDialog

Dient zur Auswahl des Funktionssimulators beim Starten des Programms oder um während der Ausführung zu einem anderen zu wechseln. Zur Auswahl stehen die verschiedenen Funktionssimulatoren.

8.1 Funktionssimulator

C FSChoiceDialog

❑ but_abort: JButton
❑ but_next: JButton
❑ connectionOptionsDialog: ConnectionOptionsDialog
❑ frame: MainFrame
❑ radioButtons: JRadioButton[]
❑ rb_Control: JRadioButton
❑ rb_IO: JRadioButton
❑ rb_IOControl: JRadioButton
❑ rb_Robot: JRadioButton
❑ rb_RobotControl: JRadioButton
❑ rb_Storage: JRadioButton
❑ rb_StorageControl: JRadioButton
❑ rb_Transport: JRadioButton
❑ rb_TransportControl: JRadioButton
§F serialVersionUID: long

● FSChoiceDialog(in owner: MainFrame, in modal: boolean, in connectionOptionsDialog: ConnectionOptionsDialog)
❑ getButtonPanel(): JPanel
❑ getRadioButtonPanel(): JPanel
❑ getSelectedRadioButton(): JRadioButton
❑ init()
❑ initButtons()
❑ positionDialog()

8.1.1.8 FSMenuBar

Sie beinhaltet die einzelnen Elemente der Menübar. Wurde am Anfang kein Modul des Funktionssimulators ausgewählt, sind nur die Elemente

- „Funktionssimulator auswählen“,
- „Ende“ und
- die Unterpunkte von „Hilfe“

aktiviert.

Neben den Menüeinträgen „Befehl“, „Modus“ und „Verbindung“ deren Unterpunkte auch in Form von Buttons im Hauptfenster vorhanden sind, gibt es weitere Funktionen. Es besteht die Möglichkeit den Funktionssimulator zu wechseln, zwischen den Ansichten (Detail- und Normalansicht) hin- und herzuschalten, die Verbindungseinstellungen (IP-Adresse, Port) zu ändern und die Hilfe aufzurufen.

8.1 Funktionssimulator

C FSMenuBar	
▫ answerPanel: AnswerPanel	
▫ connection: ConnectionPanel	
▫ help: Help	
▫ history: HistoryPanel	
▫ mi_about: JMenuItem	
▫ mi_sendern: JMenuItem	
▫ mi_automatik: JMenuItem	
▫ mi_detail_kommandoauswahl: JMenuItem	
▫ mi_detail_kommandoverlauf: JMenuItem	
▫ mi_exit: JMenuItem	
▫ mi_funktionssimulator_waehlen: JMenuItem	
▫ mi_help: JMenuItem	
▫ mi_herstellen: JMenuItem	
▫ mi_kommandoverlauf_exportieren: JMenuItem	
▫ mi_kommandoverlauf_leeren: JMenuItem	
▫ mi_manuell: JMenuItem	
▫ mi_normal_kommandoauswahl: JMenuItem	
▫ mi_normal_kommandoverlauf: JMenuItem	
▫ mi_senden: JMenuItem	
▫ mi_trennen: JMenuItem	
▫ mi_verbindungsoptionen: JMenuItem	
▫ mode: ModePanel	
SF serialVersionUID: long	
● FSMenuBar(in modePanel: ModePanel, in historyPanel: HistoryPanel, in connectionPanel: ConnectionPanel)	
● setActivated(in isActivated: boolean)	

8.1.1.9 HistoryPanel

Das HistoryPanel gibt alle ein- bzw. ausgehenden Nachrichten im Kommandoverlauf aus. Des Weiteren werden auch alle Statusmeldungen des Funktionssimulators und Fehler die während der Kommunikation auftreten darin protokolliert. Sollte die Nachricht ein Carriage Return Line Feed enthalten (wird z.B. vom Roboter angehängt), so wird dies durch zwei Pfeile dargestellt.

8.1 Funktionssimulator

C HistoryPanel

- answerPanel: AnswerPanel
- AnswersSent: int
- answersToSend: int
- but_abort: JButton
- but_clearList: JButton
- canSend: boolean
- connectionOptionsDialog: ConnectionOptionsDialog
- deleteLastLine: boolean
- higherSystem: String
- isAuto: boolean
- lastState: String
- pm_commandHistory: JPopupMenu
- §F serialVersionUID: long
- showCommands: boolean
- str_command: String
- str_desription: String
- ta_history: JTextArea
- tmplastState: String

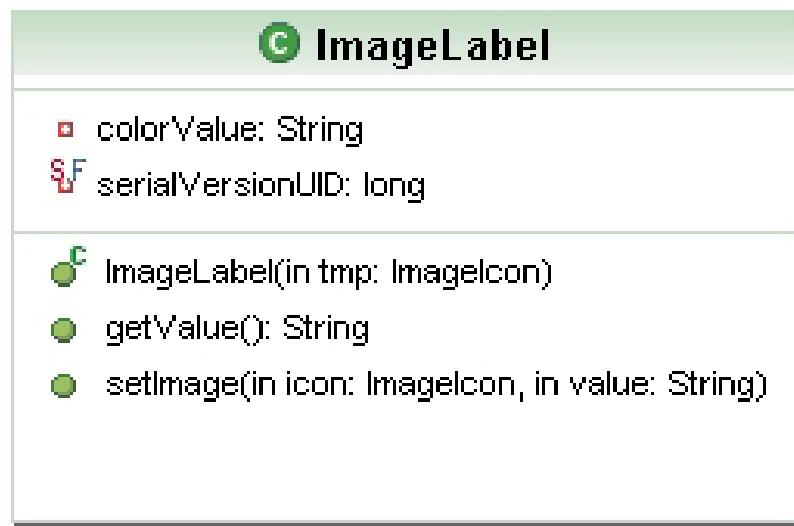
C HistoryPanel(in connectionOptionsDialog: ConnectionOptionsDialog)

- canSendCommand(): boolean
- clearAnswerPanel()
- clearHistory()
- deleteLastLine()
- getHistory(): String
- getLastState(): String
- init()
- initButton()
- initListener()
- setAnswerPanel(in answerPanel: AnswerPanel)
- setAnswersToSend(in AnswersToSend: int)
- setAutoMode(in mode: boolean)
- setCommandToHistory(in command: String, in description: String, in type: String)
- setCommandToHistory(in type: String)
- setControllingSystem(in system: String)
- setText()
- showReceivedCommands()
- showReceivedDescriptions()

8.1 Funktionssimulator

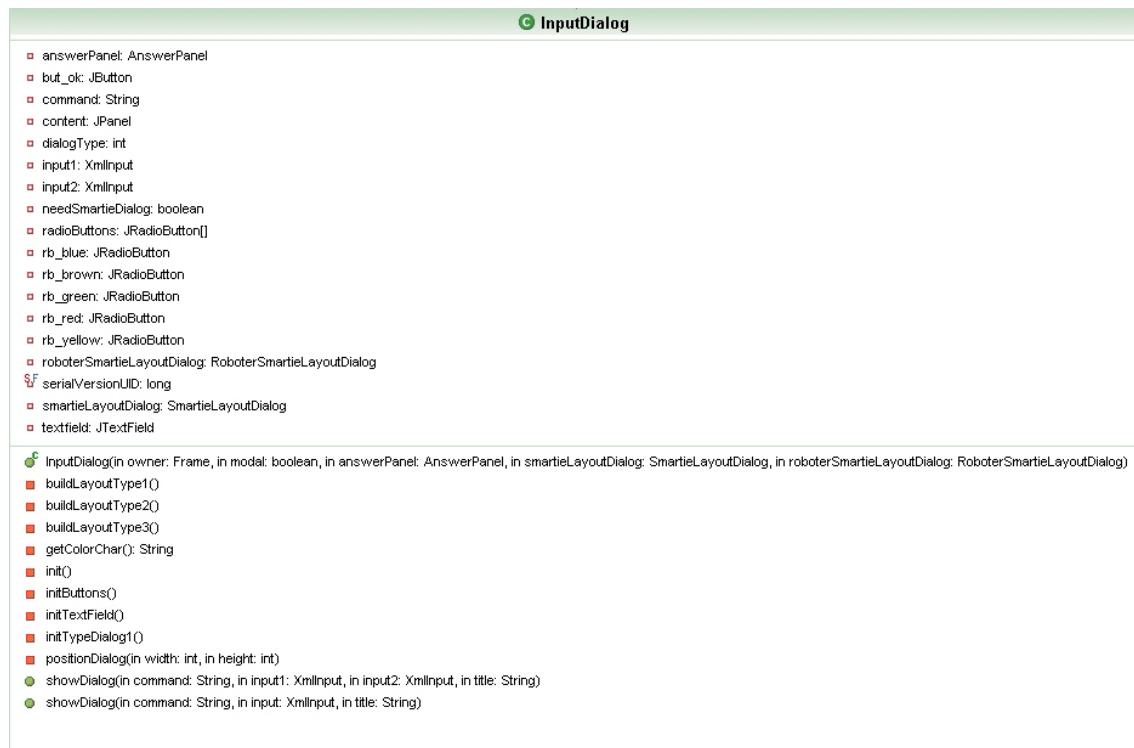
8.1.1.10 ImageLabel

Eine Klasse, die ein Bild einer Matrix-Position hält.



8.1.1.11 InputDialog

Ermöglicht es die Übergabeparameter der jeweiligen Antworten, vor dem Senden, einzustellen. Da die Antworten meist eine unterschiedliche Anzahl an Argumenten haben bzw. die Typen der Argumente verschieden sind, wird hier der jeweils richtige Dialog erstellt und angezeigt.



8.1.1.12 JSAPInputDialog

Der JSAPInputDialog ist eine andere Version des InputDialogs. Der Unterschied zum InputDialog besteht darin, dass der JSAPInputDialog einen Dialog mit 5 oder 6 Textfeldern anzeigt, während dessen der InputDialog dem Benutzer die Möglichkeit gibt eine Farbe auszuwählen und ein Textfeld zu füllen.

C JSAPInputDialog

- answerPanel: AnswerPanel
- but_ok: JButton
- colorCount: int[]
- command: String
- content: JPanel
- inputs: XmlInput
- serialVersionUID: long
- textFields: JTextField[]
- tf_blue: JTextField
- tf_brown: JTextField
- tf_green: JTextField
- tf_red: JTextField
- tf_time: JTextField
- tf_yellow: JTextField

C JSAPInputDialog(in owner: Frame, in modal: boolean, in answerPanel: AnswerPanel)

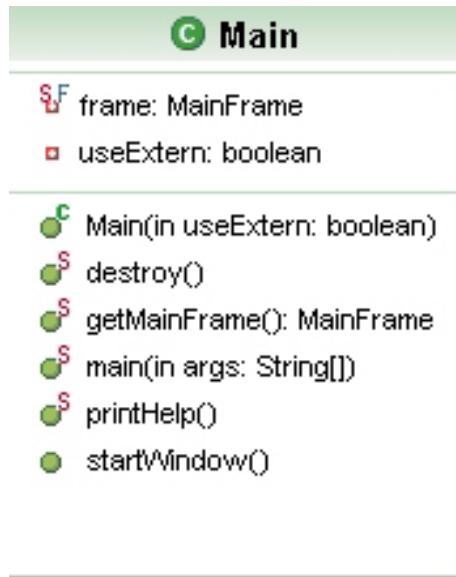
- buildLayout()
- generateDefaultParameter()
- init()
- initButton()
- initTextFields()
- positionDialog(in width: int, in height: int)
- resetDialog()
- showDialog(in command: String, in inputs: Vector, in title: String)

8.1.1.13 Main

Ist die erste Klasse die aufgerufen wird. Beim Starten überprüft sie, ob das Programm mit Argumenten aufgerufen worden ist. Ist dies der Fall, wird der Auswahldialog zum Auswählen eines

8.1 Funktionssimulator

Funktionssimulators unterdrückt und das entsprechende MainFrame initialisiert. Wird kein Parameter übergeben, wird der Funktionssimulatormauswahl dialog geöffnet.



8.1.1.14 MainFrame

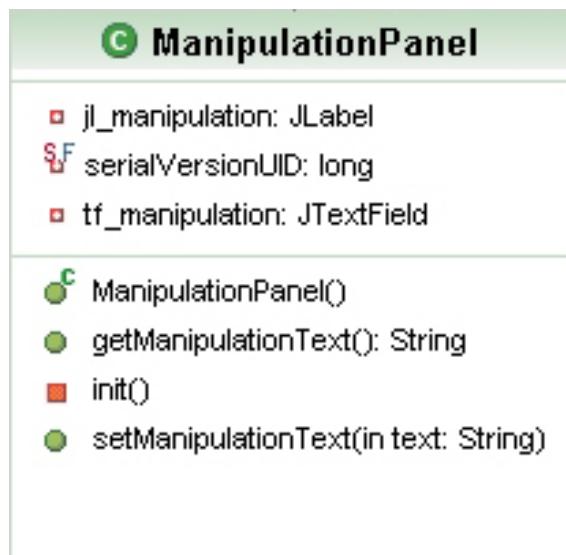
Der MainFrame ist das Hauptfenster des Programms, welches alle Elemente beinhaltet und diese je nach gestartetem Funktionssimulator richtig initialisiert.

8.1 Funktionssimulator

C MainFrame	
□	answerPanel: AnswerPanel
□	choiceDialog: FSChoiceDialog
□	connectionDialog: ConnectionDialog
□	connectionOptionsDialog: ConnectionOptionsDialog
□	connectionPanel: ConnectionPanel
□	content: JPanel
□	debug: boolean
□	debugPath: String
□	fsMenuBar: FSMenuBar
□	historyPanel: HistoryPanel
□	init: boolean
□	manipulationPanel: ManipulationPanel
□	modePanel: ModePanel
SF	serialVersionUID: long
□	useExtern: boolean
C	MainFrame()
C	MainFrame(in name: String)
R	checkArgs(in arg: String): String
G	getConnection(): ConnectionPanel
G	getFSStartString(in args: String[]): String
R	init()
G	initMainFrame(in funktionssimulator: String)
G	isDebugMode(): boolean
G	isInit(): boolean
D	processWindowEvent(in e: WindowEvent)
G	setUseExtern(in useExtern: boolean)
G	showChoiceDialog()
G	showConnectionOptionsDialog()

8.1.1.15 ManipulationPanel

Ein Panel, dass der Manipulation der Nachrichten dient.



8.1.1.16 ModePanel

Das ModePanel zeigt die zuletzt eingegangene Nachricht in einem Textfeld an und ermöglicht es, zwischen dem manuellen und dem automatischen Modus zu wechseln. Im manuellen Modus können die Antworten, die gesendet werden sollen, vom Benutzer gewählt werden, im automatischen Modus wird dies vom Programm übernommen.

C ModePanel

- answerPanel: AnswerPanel
- manipulationPanel: ManipulationPanel
- messageParser: FSMessageParser
- rb_automatic: JRadioButton
- rb_manipulate: JRadioButton
- rb_testMode: JRadioButton
- serialVersionUID: long
- tf_message: JTextField

- ModePanel(in answerPanel: AnswerPanel, in manipulationPanel: ManipulationPanel)
- changeToManipulateMode(in state: boolean)
- changeToTestModus(in wert: boolean)
- getAnswerPanel(): AnswerPanel
- getMessagePanel(): JPanel
- init()
- initComponents()
- isModeAutomatic(): boolean
- setMessage(in message: String)
- setParser(in messageParser: FSMessageParser)
- setRb_automatic()
- setRb_testMode()

8.1.1.17 nextButtonPressedHandler

Der nextButtonPressedHandler ist eng mit dem „ConnectionOptionsDialog“ verbunden, da dieser Handler aufgerufen wird, wenn bei den Verbindungsoptionen „OK“ ausgewählt wird. Nachfolgend wird die eingegebene IP-Adresse auf ihre korrekte Syntax hin überprüft. Falls sie dieser nicht entspricht, wird versucht der Inhalt des Textfeldes in eine zulässige IP-Adresse aufzulösen (z. B. „localhost“ -> 127.0.0.1). Ist danach immer noch keine Auflösung der IP-Adresse möglich, wird eine Fehlermeldung ausgegeben.

Wurden die Einstellungen auf ihre Richtigkeit überprüft, so wird die alte Verbindung getrennt und versucht, sich zu den im ConnectionOptions-Dialog eingegebenen Daten zu verbinden.

 nextButtonPressedHandler
<ul style="list-style-type: none">▫ mDialog: ConnectionOptionsDialog▫ numb_counter: int▫ point_counter: int
<ul style="list-style-type: none">● actionPerformed(in e: ActionEvent)■ isIpValid(in alp: String): boolean●  nextButtonPressedHandler(in connectionOptionsDialog: ConnectionOptionsDialog)

8.1.1.18 RoboterSmartieLayoutDialog

Der RoboterSmartieLayoutDialog ist eine Kopie des SmartieLayoutDialogs. Er wurde bei der Implementierung des Roboterssystems implementiert um auch ein nicht gelegtes Smartie als ein X zu markieren.

8.1 Funktionssimulator



8.1.1.19 SmartieLayoutDialog

Er stellt eine Lagerpalette mit der typischen Aufteilung von 13 Zeilen und 7 Spalten dar, die mit den verschiedenfarbigen Smarties bestückt werden kann. Der SmartieLayoutDialog wird benötigt, wenn für bestimmte Kommandos eine Palettenbelegung (z. B. wenn eine Lagerpalette erfolgreich ausgelagert worden ist / LaST-FS) zurückgegeben werden muss. Beim ersten Aufruf ist die Palette schon mit Default-Werten vorbelegt.

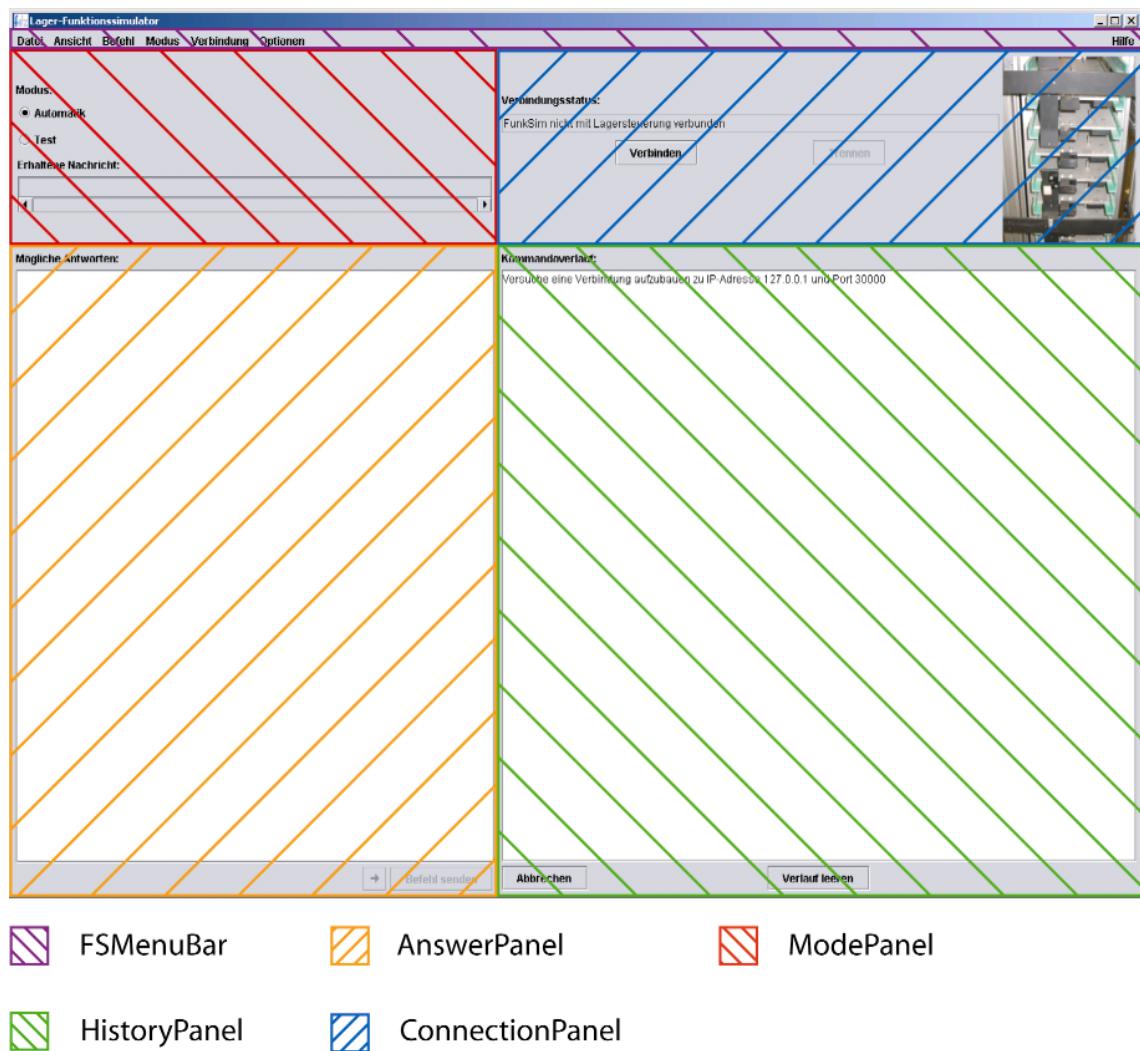
 **SmartieLayoutDialog**

- activeColor: String
- answerPanel: AnswerPanel
- bg: ButtonGroup
- but_abort: JButton
- but_ok: JButton
- buttonPanel: JPanel
- colors: Hashtable
- columns: int
- command: String
- content: JPanel
- description: String
- fc: JFileChooser
- images: Hashtable
- imgWidth: int
- input: XmlInput
- pFarben: JPanel
- pFarbenWidth: int
- palettenElemente: JLabel
- rows: int
- serialVersionUID: long

-  SmartieLayoutDialog(in owner: Frame, in modal: boolean, in answerPanel: AnswerPanel)
-  SmartieLayoutDialog(in owner: Frame, in modal: boolean, in answerPanel: AnswerPanel, in layout: String)
- actionPerformed(in e: ActionEvent)
- buildDialog()
- buildlayout(in anordnung: String)
- genButtonPanel()
- genColorPanel()
- genImageLabel()
- getPreview(): String
- init(in layout: String)
- initButtons()
- loadImages()
- posButtonPanel()
- positionDialog(in width: int, in height: int)
- setBackground()
- setColor()
- showDialog(in command: String, in description: String, in input: XmlInput)

8.1 Funktionssimulator

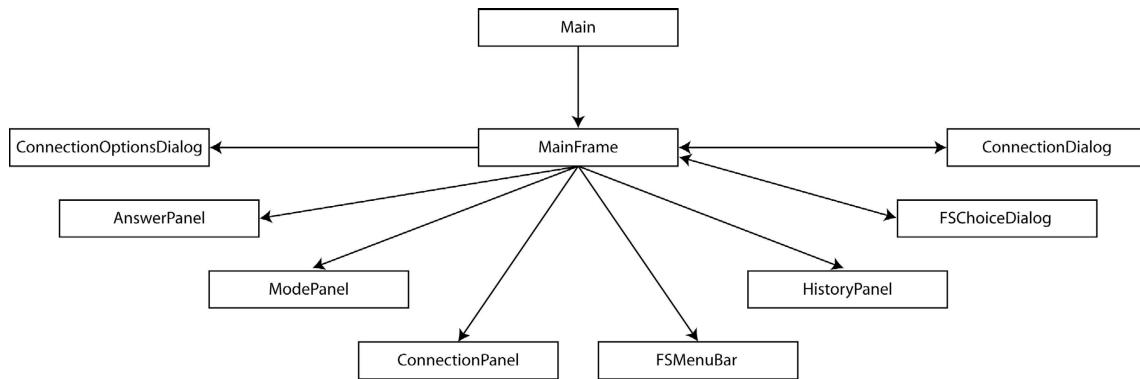
8.1.1.20 Funktionssimulator – Modularer Aufbau



Das MainFrame vom Funktionssimulator gliedert sich in verschiedene Bereiche mit unterschiedlichen Aufgaben, hinter denen jeweils eine eigene Klasse steht. Die modulare Aufteilung soll hier noch einmal etwas verdeutlicht werden.

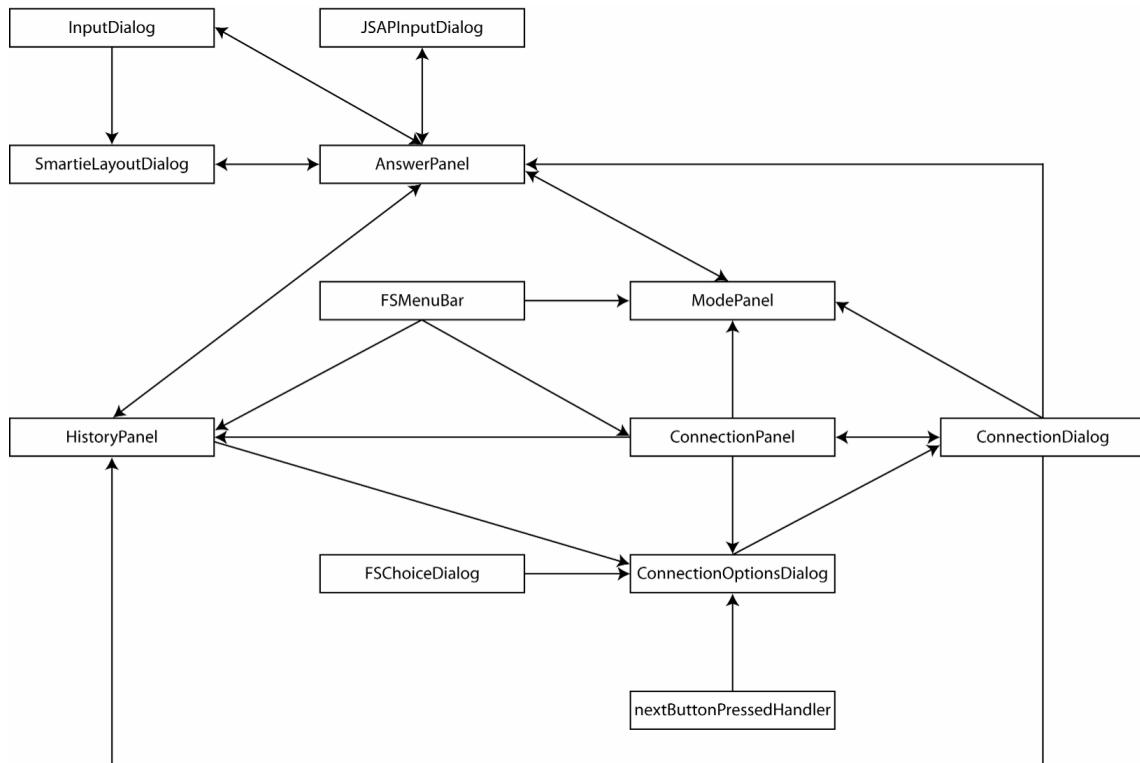
8.1 Funktionssimulator

8.1.1.20.1 presentationLayer - Klassenübersicht I



Alle Klassen die in Verbindung mit „MainFrame“ stehen. Die Klasse „Main“ ist der Einstiegspunkt in das Programm.

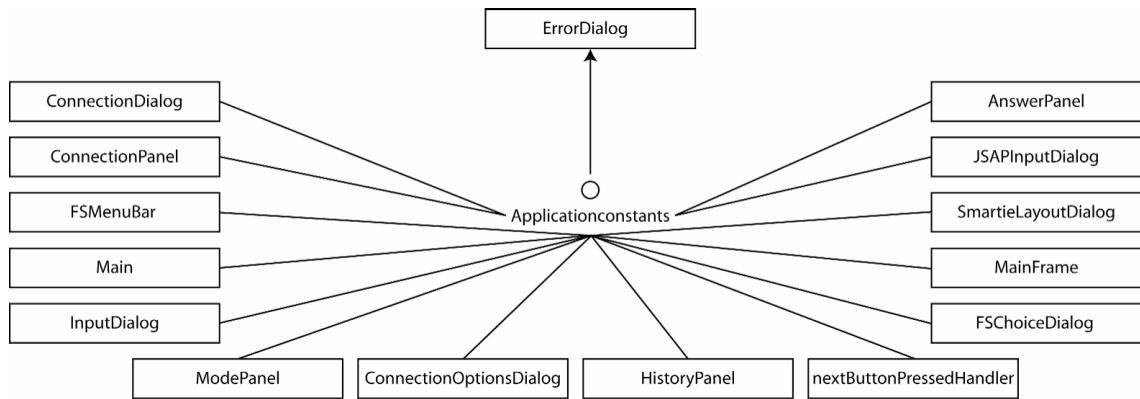
8.1.1.20.2 presentationLayer - Klassenübersicht II



Klassendiagramm, das die Verbindungen der Klassen untereinander aufzeigt, aber ohne die Klasse „MainFrame“.

8.1 Funktionssimulator

8.1.1.20.3 presentationLayer – Klassenübersicht III



Das Interface `Applicationconstants` ermöglicht es den Klassen, die es eingebunden haben, auf gemeinsame Konstanten, wie Pfade zuzugreifen und den Error Dialog aufzurufen.

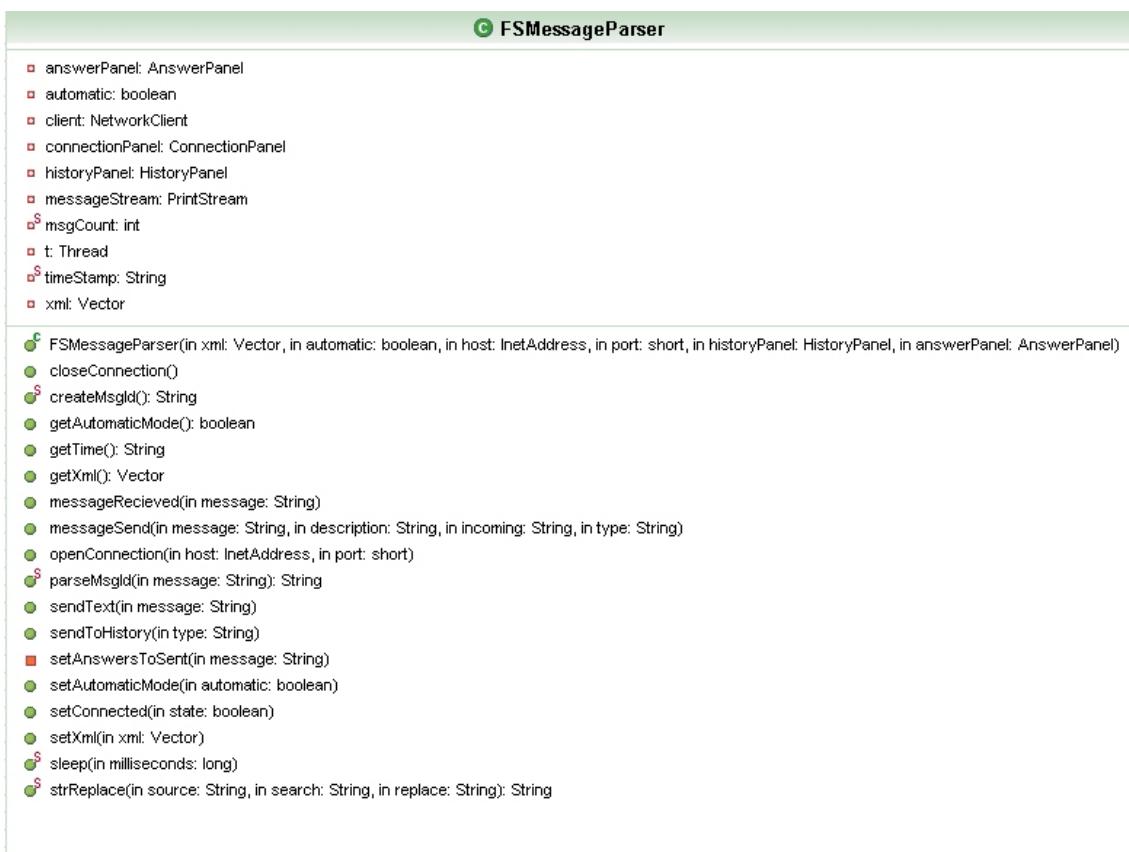
8.1.2 Package - businessLayer

Der `businessLayer` dient als Schnittstelle zwischen dem `presentationLayer` und dem `dataLayer`. Er bereitet die jeweils empfangenen Daten auf und gibt sie an den anderen Layer weiter. Dabei nutzt er für die Aufbereitung der Daten Informationen aus speziellen XML-Dateien, in denen die Kommunikationsbefehle für die verschiedenen Klassen hinterlegt sind.

8.1.2.1 FSMessageParser

Empfängt und sendet Nachrichten über die Klasse „`NetworkClient`“. Empfangene Nachrichten werden mit Hilfe der Informationen aus der entsprechenden XML-Datei überprüft und dann an die jeweilige Klassen weitergeleitet. Den zu sendenden Nachrichten wird, bevor sie an die Klasse „`NetworkServer`“ weitergeleitet werden, eine Message-ID gegeben. Außerdem wird, falls es sich um die Simulation des Roboters handelt, bei der Methode `messageSend()` ein Carriage Return Line Feed angehängt.

8.1 Funktionssimulator



8.1 Funktionssimulator

8.1.2.2 XmlModi

Parst den Inhalt der jeweiligen XML-Datei und bereitet die Daten so auf, dass sie in den verschiedenen Klassen verwendet werden können.

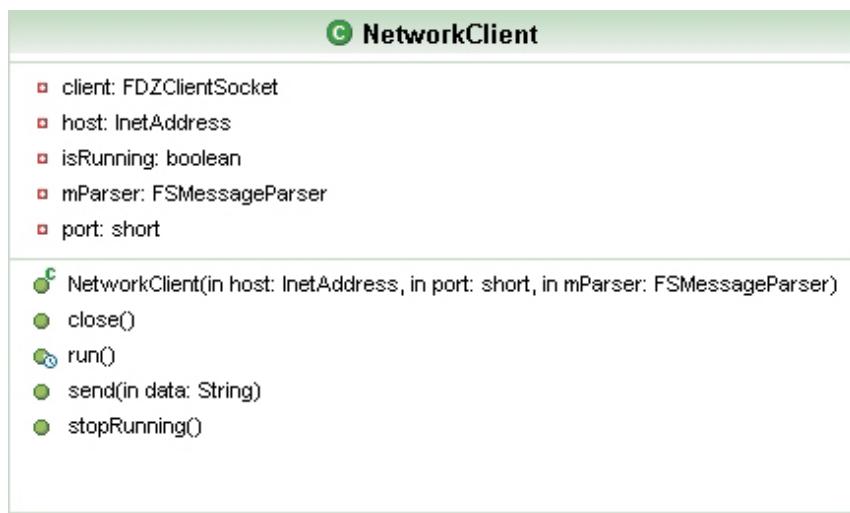


8.1.3 Package - dataLayer

Der dataLayer ist für den Datenaustausch mit der jniBridge zuständig. Daten die von der dataLayer-Schicht zum businessLayer-Schicht gehen werden dort aufbereitet und an die presentationLayer-Schicht weitergegeben.

8.1.3.1 NetworkClient

Stellt die Verbindung zwischen dem Funktionssimulator und der „jniBridge“ her. In der Klasse selbst läuft ein Thread, der auf eingehende Nachrichten von der „jniBridge“ wartet. Zu sendende Kommandos werden von hier an die „jniBridge“ weitergegeben.



8.1.3.2 StateTracer

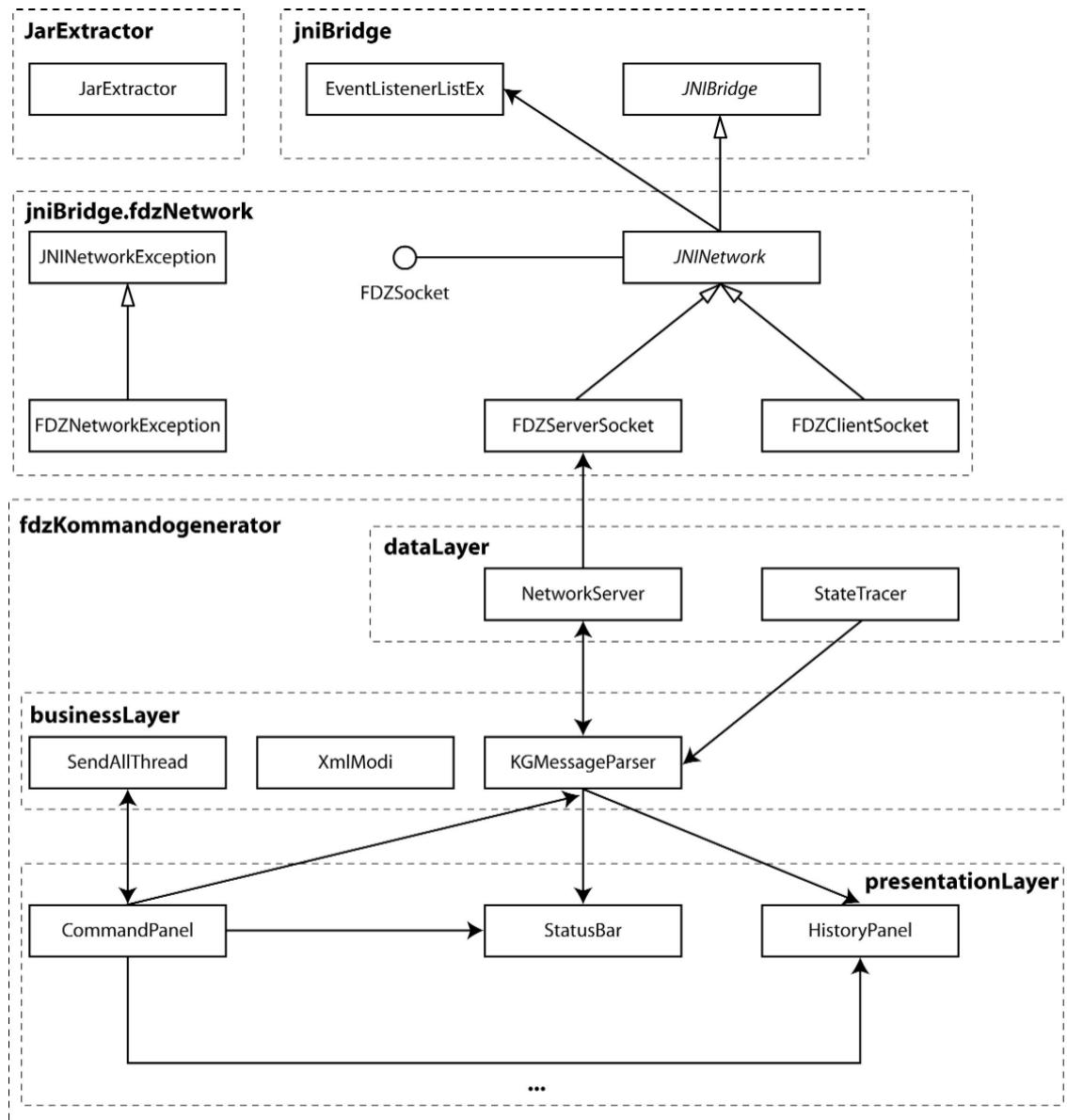
Ein Listener, der über Änderungen des Verbindungsstatus (Verbindung hergestellt bzw. verloren) informiert.



8.2 Kommandogenerator

8.2 Kommandogenerator

Der Kommandogenerator simuliert die übergeordneten FDZ-Teile, wie z.B. JSAP, Steuerung, Lagersteuerung, Mit seiner Hilfe können definierte Befehle an die untergeordneten Teile gesendet werden und die Reaktion überwacht werden.



Dies ist eine Übersicht vom Aufbau des Kommandogenerators, dem man die Verbindungen der einzelnen Klassen packageübergreifend entnehmen kann. Aus dem Package „presentationLayer“ sind nur die Klassen aufgelistet, die mit Klassen aus dem Package „businessLayer“ in Verbindung stehen. Der komplette Aufbau von „presentationLayer“ ist im entsprechenden Klassendiagramm weiter unten zu finden. Die eigentliche Verbindung zwischen Kommandogenerator und der „jni-Bridge“ wird über die Klasse „NetworkServer“ hergestellt.

8.2 Kommandogenerator

8.2.1 Package - presentationLayer

Mit dem Package presentationLayer werden die Oberflächen, für die verschiedenen FDZ-Teile, im Kommandogenerator dargestellt. Er dient der Kommunikation zwischen dem System und dem Anwender, damit entspricht er einem HMI (Human Machine Interface).

8.2.1.1 Applicationconstants

Ein Interface, das Konstanten (z. B. Pfade, Default-Ports) beinhaltet, auf welche die verschiedenen Klassen zurückgreifen können. Zusätzlich beinhaltet es auch eine Instanz des Error Dialogs, der bei aufgetretenen Fehlern aufgerufen wird und allen Klassen, die auf dieses Interface zugreifen, zur Verfügung steht.

8.2 Kommandogenerator

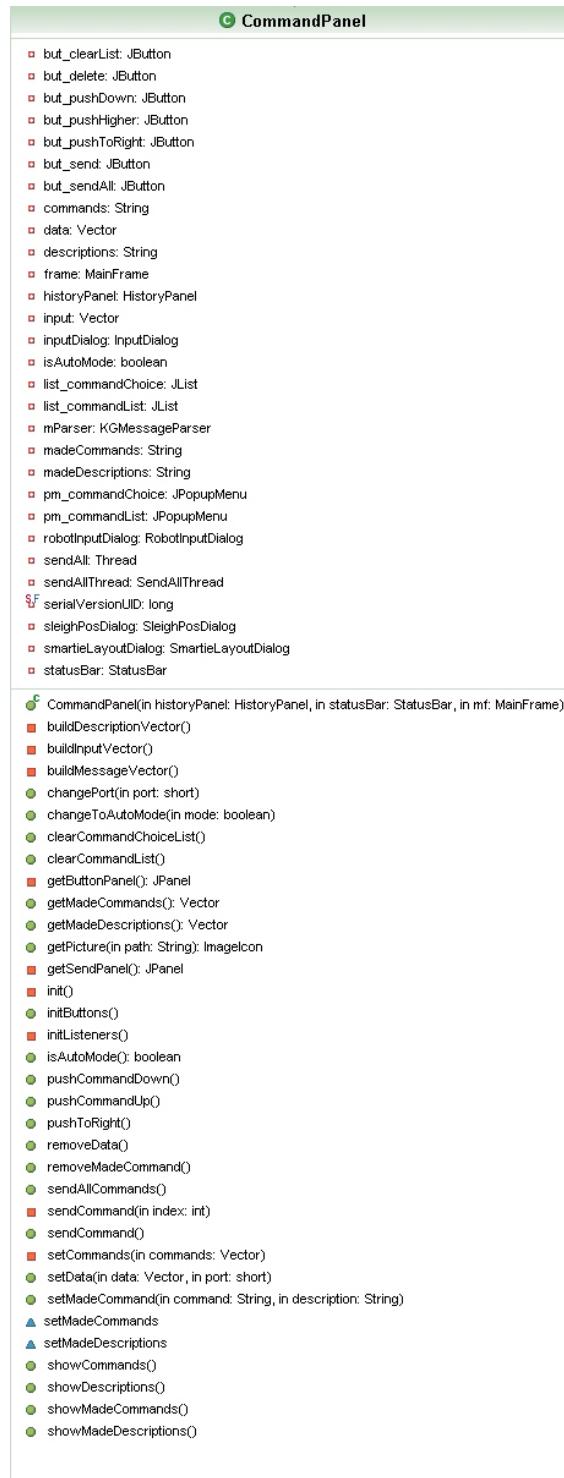
«interface»	
Applicationconstants	
\$F	CMD_ABORT: String
\$F	CMD_CONNECT: String
\$F	CMD_CONTROL: String
\$F	CMD_DISCONNECT: String
\$F	CMD_IO: String
\$F	CMD_IOCONTOL: String
\$F	CMD_JSAP: String
\$F	CMD_NEXT: String
\$F	CMD_NO: String
\$F	CMD_NUMBERS: String
\$F	CMD_OK: String
\$F	CMD_RESET: String
\$F	CMD_RGYBW: String
\$F	CMD_RGYBW_:
\$F	CMD_RGYBW_X: String
\$F	CMD_ROBOT: String
\$F	CMD_ROBOTCONTOL: String
\$F	CMD_SLEIGHPOS: String
\$F	CMD_STORAGE: String
\$F	CMD_STORAGECONTROL: String
\$F	CMD_TRANSPORT: String
\$F	CMD_TRANSPORTCONTOL: String
\$F	CMD_YES: String
\$F	DEBUG: String
\$F	DEBUG_PATH: String
\$F	DEFAULT_IP: String
\$F	DEFAULT_PORT: String
\$F	DEFAULT_PORT_CO: String
\$F	DEFAULT_PORT_IO: String
\$F	DEFAULT_PORT_JOCO: String
\$F	DEFAULT_PORT_RO: String
\$F	DEFAULT_PORT_ROCO: String
\$F	DEFAULT_PORT_ST: String
\$F	DEFAULT_PORT_STCO: String
\$F	DEFAULT_PORT_TR: String
\$F	DEFAULT_PORT_TRCO: String
\$F	defaultToolkit: Toolkit
\$F	errorDialog: ErrorDialog
\$F	FILE_CONNECTION: String
\$F	FILE_IO_XML: String
\$F	FILE_IO_XML_EXTERN: String
\$F	FILE_JSAP_XML: String
\$F	FILE_JSAP_XML_EXTERN: String
\$F	FILE_MESSAGE_STREAM: String
\$F	FILE_ROBOT_XML: String
\$F	FILE_ROBOT_XML_EXTERN: String
\$F	FILE_STORAGE_XML: String
\$F	FILE_STORAGE_XML_EXTERN: String
\$F	FILE_TRANSPORT_XML: String
\$F	FILE_TRANSPORT_XML_EXTERN: String
\$F	ICON_CONTROL: String
\$F	ICON_ICON: String
\$F	ICON_INFO: String
\$F	ICON_IO: String
\$F	ICON_RIGHT: String
\$F	ICON_ROBOT: String
\$F	ICON_STORAGE: String
\$F	ICON_TRANSPORT: String
\$F	JAR_NAME: String
\$F	jarExtractor: JarExtractor
\$F	OUT_ABORT: String
\$F	OUT_CONNECTED: String
\$F	OUT_DATA_CHANGED: String
\$F	OUT_ERROR: String
\$F	OUT_ERROR_UNKNOWN: String
\$F	OUT_LOST: String
\$F	OUT_MANUALLY_DISCONNECTED: String
\$F	OUT_READY: String
\$F	OUT_RECEIVED: String
\$F	OUT_SENT: String
\$F	OUT_TRY: String
\$F	OUT_WAITING: String
\$F	OUT_WAITING_CON: String
\$F	XML_TYPE_ERROR: String
\$F	XML_TYPE_SUCCESS: String

8.2 Kommandogenerator

8.2.1.2 CommandPanel

Das CommandPanel ist für das gesamte Handling der Kommandos zuständig. D. h. es holt sich für den jeweils eingestellten Kommandogenerator die Kommandos, die gesendet werden können, zeigt diese in der Kommandoauswahlliste an und ruft, wenn nötig, den „InputDialog“ auf. über den InputDialog können vor dem Senden die Übergabeparameter dem CommandPanel mitgeteilt werden, damit die Nachrichten entsprechend korrekt erstellt werden können. Außerdem listet es die zum Senden bereiten Nachrichten in der Kommandoliste auf. Hier kann die Sende-reihenfolge noch vom Benutzer verändert werden. Auch können Kommandos gelöscht werden. Zusätzlich ermöglicht es die Auswahl zwischen dem manuellen bzw. automatischen Senden der Nachrichten.

8.2 Kommandogenerator



8.2.1.3 ConnectionOptionsDialog

Dieser Dialog ist über den Menüpunkt „Optionen“ zu erreichen und ermöglicht es die eigene IP-Adresse anzusehen und den Port umzustellen, auf dem nach eingehenden Verbindungen gehört werden soll. Die jeweils zuletzt gemachte Einstellung wird in einer Datei gespeichert und beim nächsten Start des jeweiligen Kommandogenerators wieder übernommen. Wird die Datei gelöscht bzw. ist sie beim Starten nicht vorhanden, dann wird der in der Spezifikation festgelegte Port genommen.

C ConnectionOptionsDialog

- △ but_abort: JButton
- △ but_ok: JButton
- △ connectionDialog: ConnectionDialog
- ▣ connection_properties: Properties
- § F serialVersionUID: long
- △ simulator: String
- △ st_ip: String
- △ st_port: String
- △ st_tmplIP: String
- △ st_tmpPort: String
- △ tf_ip: JTextField
- △ tf_port: JTextField

- C ConnectionOptionsDialog(in owner: MainFrame, in modal: boolean)
- getIP(): String
- getPort(): String
- init()
- initButtons()
- loadConnectionProperties()
- positionDialog()
- saveConnectionProperties()
- setConnectionDialog(in connectionDialog: ConnectionDialog)
- setPort(in simulator: String)
- showDialog(in simulator: String)

8.2.1.4 **ErrorDialog**

Erstellt für aufgetretene Fehler den richtigen Dialog. D.h. kreiert einen Dialog mit dem jeweiligen Fehlertext und der daraus entstehenden Möglichkeit darauf zu reagieren.

C **ErrorDialog**

■ but_no: JButton
■ but_ok: JButton
■ but_yes: JButton
■ content: JPanel
■ currentDialog: int
■ lb_error: JLabel
S F ■ serialVersionUID: long
■ sp_error: JScrollPane
■ tp_error: JTextPane
C ErrorDialog(<i>in owner: JFrame, in modal: boolean</i>)
● actionPerformed(<i>in e: ActionEvent</i>)
■ getButtonPanel(): JPanel
■ init()
■ positionDialog()
● showErrorDialog(<i>in title: String, in errorMessage: String</i>)
● showErrorDialog(<i>in errorMessage: String</i>)
● showFatalErrorDialog(<i>in errorMessage: String</i>)
● showFatalErrorDialog(<i>in title: String, in errorMessage: String</i>)

8.2 Kommandogenerator

8.2.1.5 HistoryPanel

Gibt alle ein- und ausgehenden Nachrichten im Kommandooverlauf aus. Dazu gehören auch alle Statusmeldungen des Kommandogenerators und die Fehler die während der Kommunikation aufgetreten sind. Des Weiteren überprüft er, ob die Nachrichten in der richtigen Reihenfolge eingetroffen sind, um dann in den jeweiligen Status zu gehen. Sollte die Nachricht ein Carriage Return Line Feed enthalten, wird z.B. vom Roboter angehängt, so wird dies durch zwei Pfeile dargestellt.

8.2 Kommandogenerator

C HistoryPanel

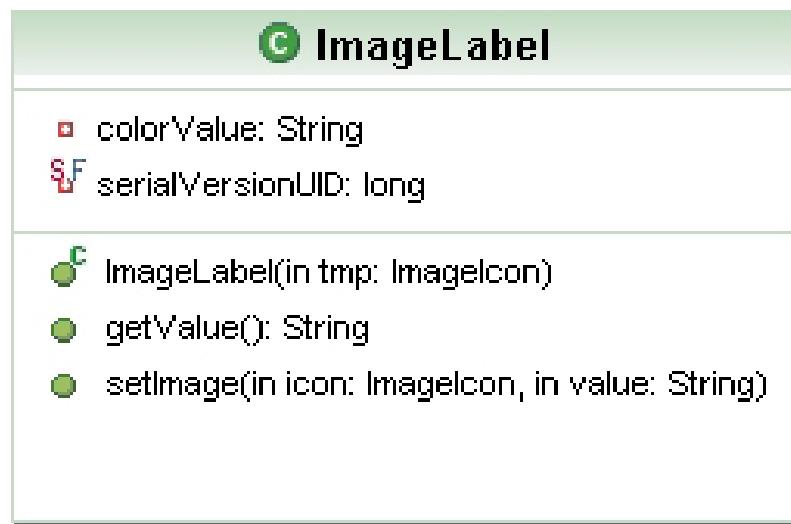
- answerPanel: AnswerPanel
- AnswersSent: int
- answersToSend: int
- but_abort: JButton
- but_clearList: JButton
- canSend: boolean
- connectionOptionsDialog: ConnectionOptionsDialog
- deleteLastLine: boolean
- higherSystem: String
- isAuto: boolean
- lastState: String
- pm_commandHistory: JPopupMenu
- §F serialVersionUID: long
- showCommands: boolean
- str_command: String
- str_desription: String
- ta_history: JTextArea
- tmplastState: String

C HistoryPanel(in connectionOptionsDialog: ConnectionOptionsDialog)

- canSendCommand(): boolean
- clearAnswerPanel()
- clearHistory()
- deleteLastLine()
- getHistory(): String
- getLastState(): String
- init()
- initButton()
- initListener()
- setAnswerPanel(in answerPanel: AnswerPanel)
- setAnswersToSend(in AnswersToSend: int)
- setAutoMode(in mode: boolean)
- setCommandToHistory(in command: String, in description: String, in type: String)
- setCommandToHistory(in type: String)
- setControllingSystem(in system: String)
- setText()
- showReceivedCommands()
- showReceivedDescriptions()

8.2.1.6 ImageLabel

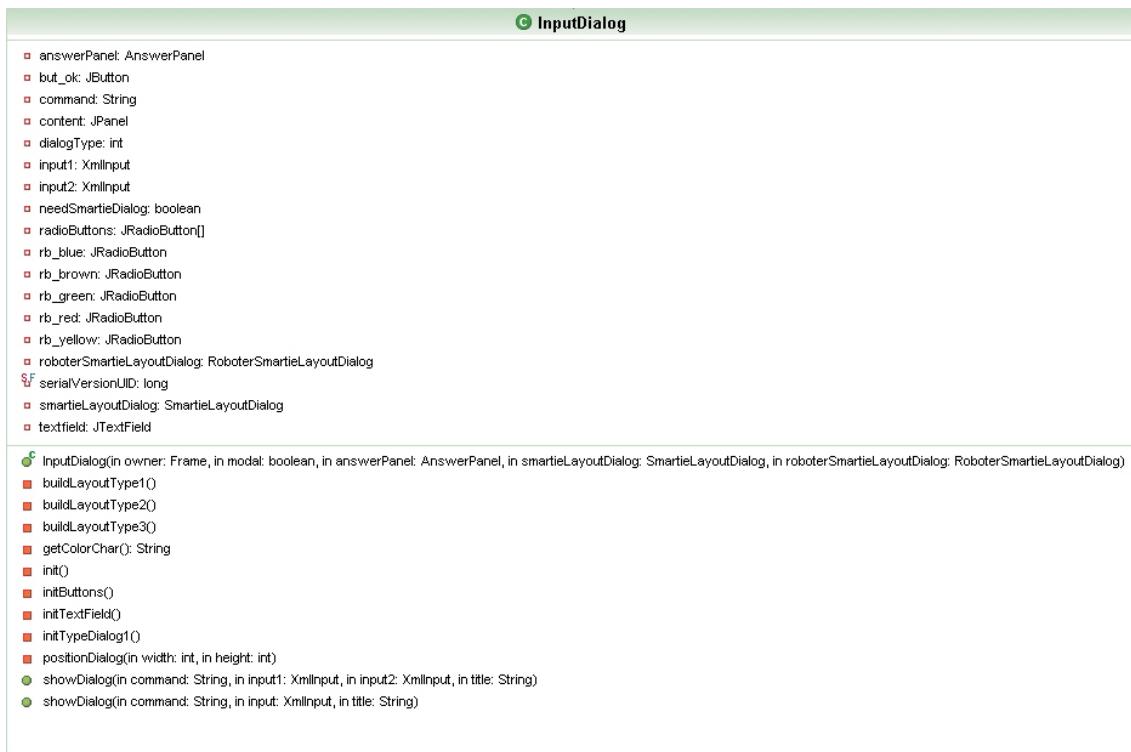
Ein Instanz der InputDialog Klasse erzeugt ein JPanel auf dem eine Paletten-Position dargestellt wird. Objekte dieser Klasse werden nur für den SmartieLayoutDialog verwendet.



8.2.1.7 InputDialog

Ermöglicht es die Übergabeparameter der jeweiligen Kommandos, vor dem Einfügen aus der Kommandoauswahlliste in die Kommandoliste, einzustellen. Da die zu sendenden Kommandos meist eine unterschiedliche Anzahl an Argumenten haben bzw. die Typen der Argumente verschieden sind, wird hier der jeweils richtige Dialog erstellt und angezeigt.

8.2 Kommandogenerator



8.2.1.8 KGChoiceDialog

Dient zur Auswahl des Kommandogenerators beim Starten des Programms oder um während der Ausführung zu einem anderen zu wechseln. Zur Auswahl stehen die verschiedenen Kommandogeneratoren.

C KGChoiceDialog

- ❑ but_abort: JButton
- ❑ but_next: JButton
- ❑ frame: MainFrame
- ❑ radioButtons: JRadioButton[]
- ❑ rb_Control: JRadioButton
- ❑ rb_IOContol: JRadioButton
- ❑ rb_JSAP: JRadioButton
- ❑ rb_RobotControl: JRadioButton
- ❑ rb_StorageControl: JRadioButton
- ❑ rb_TransportControl: JRadioButton
- ❑ serialVersionUID: long

C KGChoiceDialog(in owner: MainFrame, in modal: boolean)

- ❑ getButtonPanel(): JPanel
- ❑ getRadioButtonPanel(): JPanel
- ❑ getSelectedRadioButton(): JRadioButton
- ❑ init()
- ❑ initButtons()
- ❑ positionDialog()

8.2.1.9 KGMenuBar

Sie beinhaltet die einzelnen Elemente der Menübar. Wurde am Anfang kein spezielles Modul des Kommandogenerators ausgewählt, sind nur die Elemente

- „Kommandogenerator auswählen“,
- „Ende“ und
- die Unterpunkte von „Hilfe“

aktiviert.

Neben den Unterpunkten des Menüeintrags „Befehl“, die auch in Form von Buttons im Hauptfenster vorhanden sind, gibt es weitere Funktionen. Es besteht die Möglichkeit den Kommandogenerator zu wechseln, den Inhalt der Kommandoliste zu speichern/laden und den Kommandoverlauf

8.2 Kommandogenerator

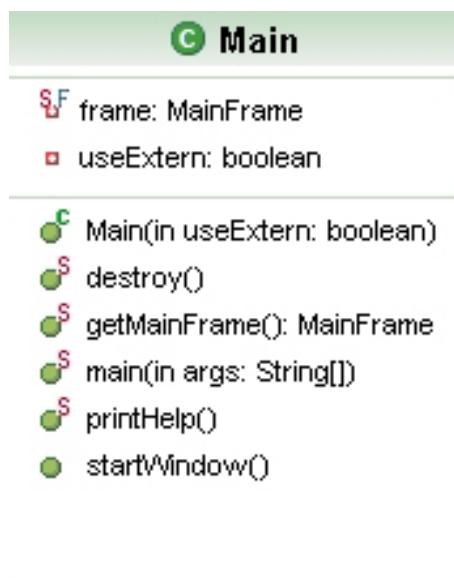
zu sichern. Des Weiteren ist es möglich zwischen den Ansichten (Detail- und Normalansicht) hin- und herzuschalten, die Verbindungseinstellungen (Port) zu ändern und die Hilfe aufzurufen.



8.2.1.10 Main

Ist die erste Klasse die aufgerufen wird. Überprüft ob das Programm mit Argumenten aufgerufen worden ist. Ist dies der Fall, wird der Auswahldialog zum Auswählen eines Kommandogenerators unterdrückt und das MainFrame direkt initialisiert. Bei keinen übergebenen Parametern, wird der Kommandogenerator-Auswahldialog angezeigt.

8.2 Kommandogenerator



8.2.1.11 MainFrame

Sozusagen das Hauptfenster des Programms, welches alle Elemente beinhaltet und sie je nach gestartetem Kommandogenerator richtig initialisiert.

Erstellt, wenn der Steuerungs-Kommandogenerator ausgewählt worden ist, zusätzlich zu den üblichen Komponenten, eine Combo-Box mit den jeweiligen Sub-Systemen die sich zum Kommandogenerator verbinden können. Wird eines ausgewählt und ist dieses auch verbunden, so wird die dazu gehörige XML-Datei mit den möglichen Kommandos geladen.

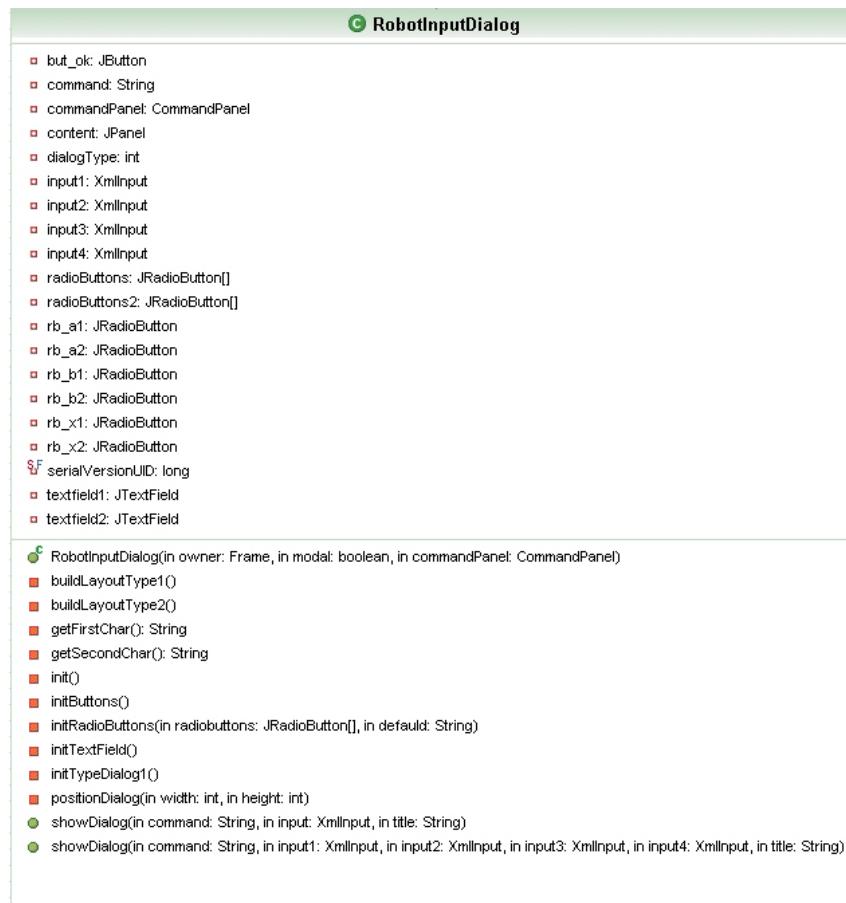
8.2 Kommandogenerator



8.2.1.12 RobotInputDialog

Der RobotInputDialog ist eine Abwandlung des InputDialogs. Er wird angezeigt, wenn ein Befehl beim Kommandogenerator Robotersteuerung erzeugt wird. Er wurde bei der Implementation des Robotersystems erstellt.

8.2 Kommandogenerator



8.2.1.13 SleighPosDialog

Stellt einen Schlittenpositionsdialog dar. Zur Auswahl stehen die drei Schlittenpositionen:

- ro
- la
- ea

8.2 Kommandogenerator



8.2.1.14 SmartieLayoutDialog

Er stellt eine Produktpalette mit der typischen Aufteilung von 9 Zeilen und 7 Spalten, bzw. eine Lagerpalette mit 13 Zeilen und 7 Spalten dar. Diese können mit den verschiedenfarbigen Smarties (Rot, Gelb, Grün, Blau, Braun) bestückt werden. Er wird benötigt, wenn für manche Kommandos eine Palettenbelegung (z. B. „Start eines Bestückungsvorgangs“ / JSAP-KG) übergeben werden muss. Beim Erscheinen ist die Palette schon mit Default-Werten vorbelegt.

C SmartieLayoutDialog

- activeColor: String
- answerPanel: AnswerPanel
- bg: ButtonGroup
- but_abort: JButton
- but_ok: JButton
- buttonPanel: JPanel
- colors: Hashtable
- columns: int
- command: String
- content: JPanel
- description: String
- fc: JFileChooser
- images: Hashtable
- imgWidth: int
- input: XmlInput
- pFarben: JPanel
- pFarbenWidth: int
- palettenElemente: JLabel
- rows: int
- serialVersionUID: long

- SmartieLayoutDialog(in owner: Frame, in modal: boolean, in answerPanel: AnswerPanel)
- SmartieLayoutDialog(in owner: Frame, in modal: boolean, in answerPanel: AnswerPanel, in layout: String)
- actionPerformed(in e: ActionEvent)
- buildDialog()
- buildlayout(in anordnung: String)
- genButtonPanel()
- genColorPanel()
- genImageLabel()
- getPreview(): String
- init(in layout: String)
- initButtons()
- loadImages()
- posButtonPanel()
- positionDialog(in width: int, in height: int)
- setBackground()
- setColor()
- showDialog(in command: String, in description: String, in input: XmlInput)

8.2 Kommandogenerator

8.2.1.15 StatusBar

Zeigt den Verbindungsstatus zum Funktionssimulator an. Dabei wird immer der Name des aktuell gestarteten Kommandogenerator, der Name des Funktionssimulators mit dem er verbunden sein müsste und der momentane Status der Verbindung (verbunden / nicht verbunden) angezeigt.

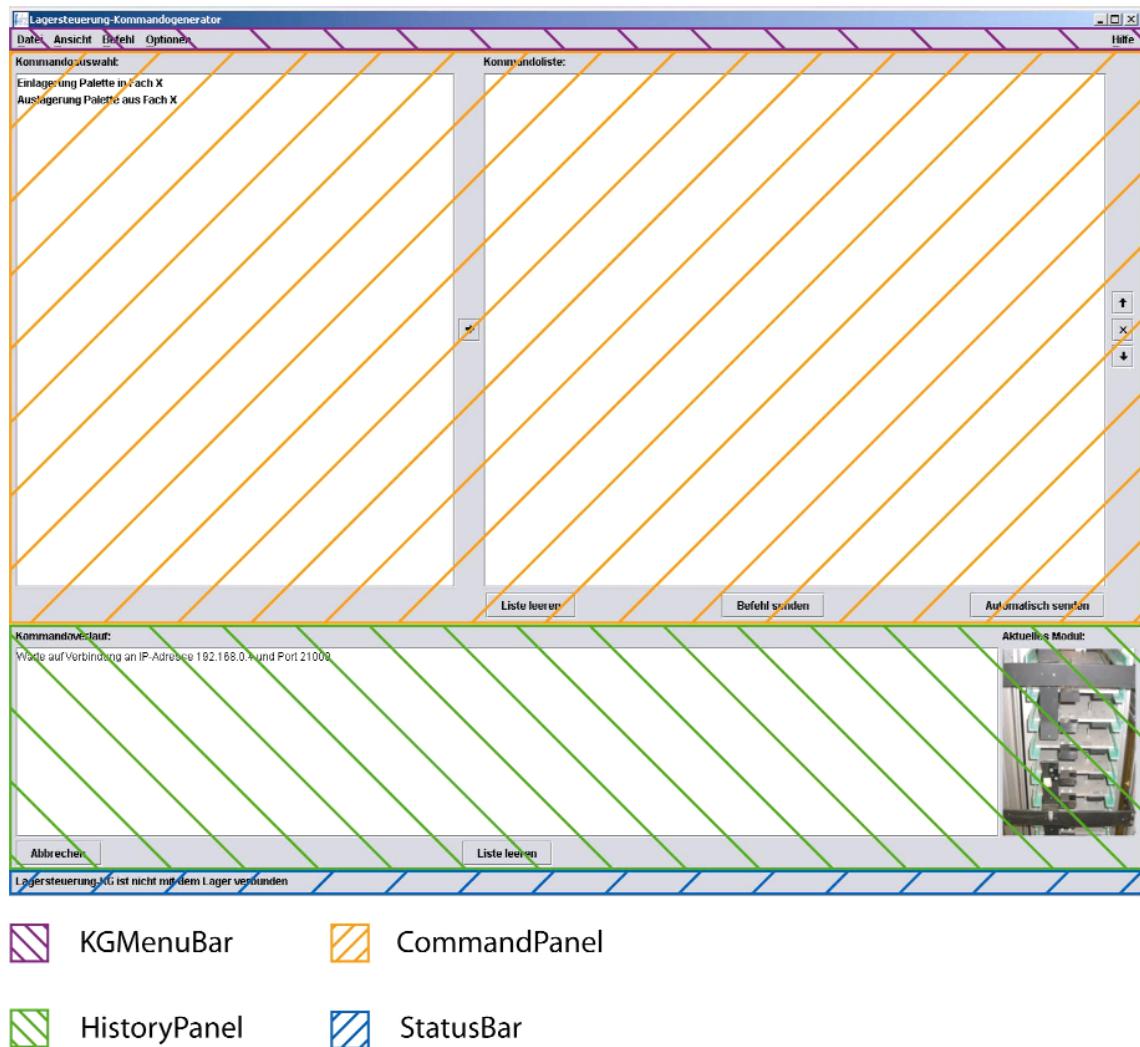
C StatusBar

- connection: boolean
- controllingSystem: String
- lb_connection: JLabel
- § F serialVersionUID: long
- subSystem: String

- C StatusBar()
- isConnected(): boolean
- setConnection(in status: boolean)
- setControllingSystem(in system: String, in subSystem: String)
- setControllingSystem(in system: String)

8.2 Kommandogenerator

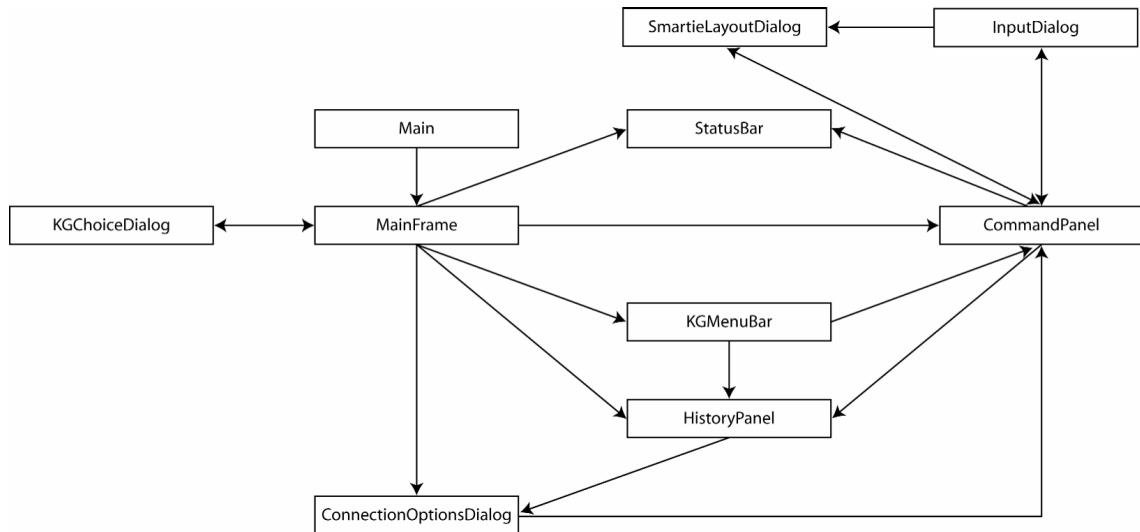
8.2.1.16 Modularer Aufbau



Das MainFrame vom Kommandogenerator gliedert sich in verschiedene Bereiche mit unterschiedlichen Aufgaben, hinter denen jeweils eine eigene Klasse steht. Die modulare Aufteilung soll hier noch einmal verdeutlicht werden.

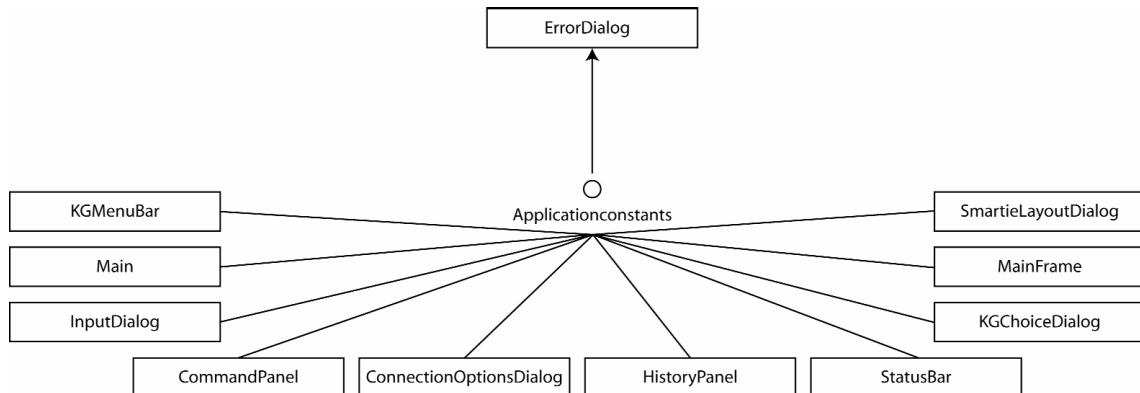
8.2 Kommandogenerator

8.2.1.16.1 presentationLayer – Klassenübersicht I



Klassendiagramm, das die Verbindungen der Klassen untereinander aufzeigt. Das Programm wird mit der Klasse „Main“ gestartet. Die eigentlichen Hauptklassen sind aber „MainFrame“ und „CommandPanel“. Während „MainFrame“ sämtliche Module miteinander verbindet und das Hauptfenster darstellt, steckt in „CommandPanel“ die gesamte Logik im Umgang mit den Nachrichten.

8.2.1.16.2 presentationLayer – Klassenübersicht II



Das Interface Applicationconstants ermöglicht es den Klassen, die es eingebunden haben, auf gemeinsame Konstanten, wie Pfade zuzugreifen und den Error Dialog aufzurufen.

8.2.2 Package - businessLayer

Der businessLayer dient als Schnittstelle zwischen dem presentationLayer und dem dataLayer. Er bereitet die jeweils empfangenen Daten auf und gibt sie an den anderen Layer weiter. Dabei

8.2 Kommandogenerator

nutzt er für die Aufbereitung der Daten Informationen aus speziellen XML-Dateien, in denen die Kommunikationsbefehle für die verschiedenen Klassen hinterlegt sind.

8.2.2.1 KGMessageParser

Empfängt und sendet Nachrichten über die Klasse „NetworkServer“. Empfangene Nachrichten werden mit Hilfe der Daten in der XML-Datei überprüft und dann an die entsprechenden Klassen weitergeleitet. Den zu sendenden Nachrichten wird, bevor sie an die Klasse „NetworkServer“ weitergeleitet werden, eine Message-ID gegeben.

C KGMessageParser	
□	historyPanel: HistoryPanel
□	messageStream: PrintStream
▣ S	msgCount: int
□	server: NetworkServer
□	statusBar: StatusBar
□	t: Thread
▣ S	timeStamp: String
□	xml: Vector
▣ C	KGMessageParser(in xml: Vector, in port: short, in historyPanel: HistoryPanel, in statusBar: StatusBar)
●	closeConnection()
▣ S	createMsgId(): String
●	getTime(): String
●	getXml(): Vector
●	messageRecieved(in message: String)
●	messageSend(in message: String, in description: String): boolean
▣ S	parseMsgId(in message: String): String
●	sendMessageToHistoryPanel(in type: String)
▣	setAnswersToReceive(in message: String)
●	setConnected(in state: boolean)
●	setXml(in xml: Vector)
▣ S	sleep(in milliseconds: long)
●	strReplace(in source: String, in search: String, in replace: String): String

8.2 Kommandogenerator

8.2.2.2 SendAllThread

Diese Klasse wird für den Automatik-Modus des Kommandogenerators benötigt, in welchem die Kommandos selbstständig gesendet werden. Dazu wird in diesem Thread die notwendige Methode des „CommandPanels“ aufgerufen.



8.2 Kommandogenerator

8.2.2.3 XmlModi

Parst den Inhalt der jeweiligen XML-Datei und bereitet die Daten so auf, dass sie in den verschiedenen Klassen verwendet werden können.



8.2.3 Package - dataLayer

Der dataLayer ist für den Datenaustausch mit der jniBridge zuständig. Daten die von der dataLayer-Schicht zum businessLayer-Schicht gehen werden hier mit Hilfe der Informationen aus den Steuerungsdateien (XML-Dateien) aufbereitet und an die presentationLayer-Schicht weitergegeben.

8.2.3.1 NetworkServer

Stellt die Verbindung zwischen dem Kommandogenerator und der „jniBridge“ her. In der Klasse selbst läuft ein Thread, der auf eingehende Nachrichten von der „jniBridge“ wartet. Zu sendende Kommandos werden von hier an die „jniBridge“ weitergegeben.

C NetworkServer

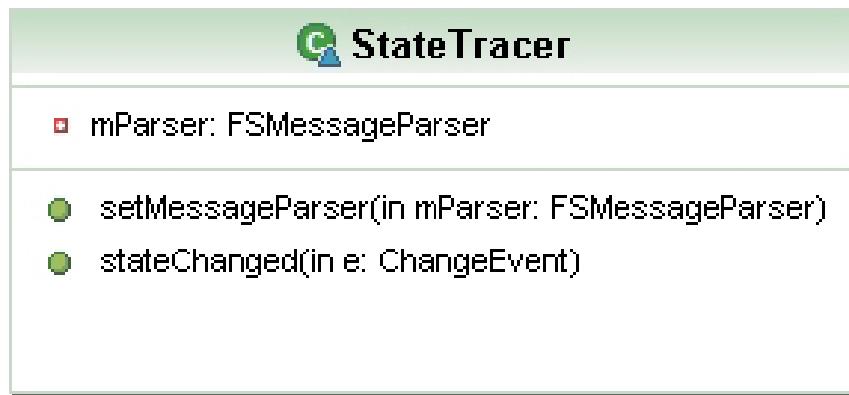
- isRunning: boolean
- mParser: KGMessageParser
- port: short
- server: FDZServerSocket

- NetworkServer(in port: short, in mParser: KGMessageParser)
- close()
- run()
- send(in data: String)
- stopRunning()

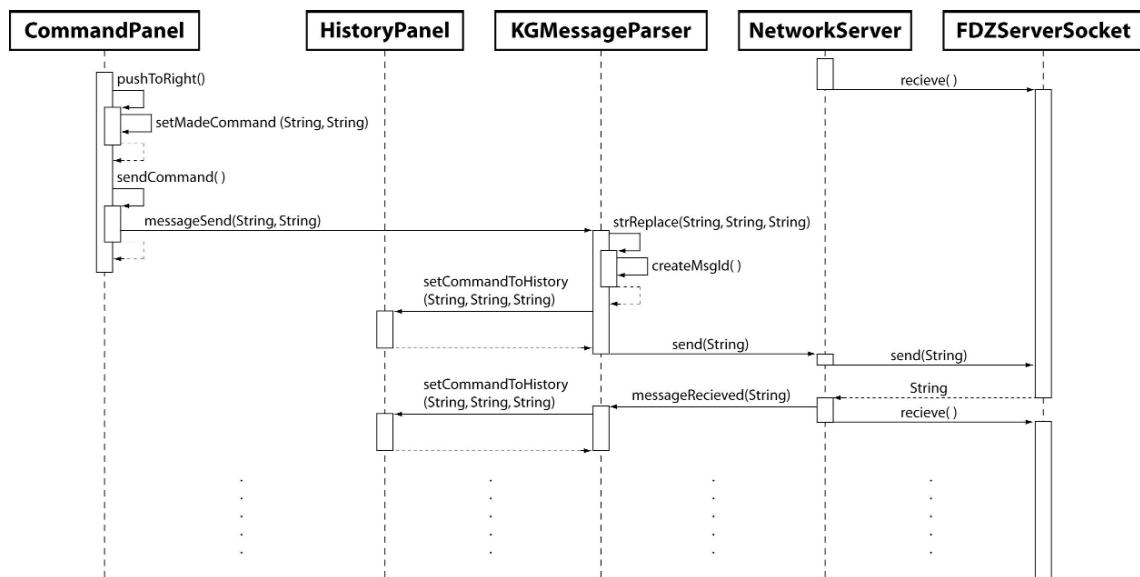
8.2.3.2 StateTracer

Ein Listener, der über Änderungen des Verbindungsstatus (Verbindung hergestellt bzw. verloren) informiert.

8.2 Kommandogenerator



8.2.3.3 Sendevorgang



Zum Senden eines Kommandos muss dieses erst zur Kommandoliste hinzugefügt werden. Dies geschieht nachdem ein Kommando in der Kommandoauswahlliste ausgewählt und der Button  geklickt worden ist. Dies veranlasst, dass im Command Panel die Methode „pushToRight()“ aufgerufen wird. Darin wird, falls dem Kommando Übergabeparameter mitgegeben werden müssen, der Input Dialog aufgerufen. Dies ist in diesem Beispiel nicht der Fall. Hier kann das Kommando sofort im Command Panel mit der Methode „setMadeCommand()“ erstellt und der Kommandoliste hinzugefügt werden, da keine Argumente notwendig sind.

Markiert der Benutzer dann einen Befehl in der Kommandoliste und klickt auf „Befehl senden“, wird die Methode „messageSend()“ im KGMessageParser aufgerufen, die dem Kommando mit Hilfe der Methoden „strReplace()“ und „createMessageID()“ eine aktuelle Message-ID hinzufügt.

Danach wird die fertig gestellte Nachricht über die Methode „`setCommandToHistory()`“ im Kommandoverlauf ausgegeben und mit der Methode „`send()`“ der Klasse `NetworkServer` gesendet.

8.3 Die Log-Datei

In der Klasse NetworkServer läuft, sobald sich ein Client zum Kommandogenerator verbunden hat ein Thread, der auf eingehende Nachrichten wartet indem die Methode „receive()“ des FDZ-ServerSockets aufgerufen wird. Aus dieser wird nur kurz zurückgekehrt, wenn eine Nachricht empfangen worden ist und dabei diese als String zurückgegeben. Danach wird nachdem die Nachricht über die Methdode „messageReceived()“ an den KGMessageParser weitergeleitet worden ist, sofort durch den Thread wieder die Methode „receive()“ aufgerufen, die dann wieder auf eingehende Nachrichten wartet.

Der KGMessageParser gibt währenddessen die empfangende Nachricht, nachdem sie intern geprüft worden ist, mit der Methode „setCommandToHistory()“ im Kommandooverlauf aus.

8.3 Die Log-Datei

Eine kleine Übersicht von Fehlern und ihrer Bedeutung, die in der Log-Datei des jeweiligen Simulationssystems auftauchen können.

Fehler die in beiden Systemen auftreten können:

java.lang.UnsatisfiedLinkError: no jniBridge in java.library.path

Möglicherweise ist die jniBridge.dll nicht im src-Verzeichnis vorhanden.

IOException in JarExtractor ...

Es ist möglicherweise keine fdzKommandogenerator.jar erstellt worden.

Error during receiving a message: 10054

Während auf eine Nachricht gewartet wird, ist ein Fehler aufgetreten, z. B. die Verbindung wurde getrennt.

Kommandogenerator:

Error during waiting for incomming connection: 10004

Erscheint, wenn ein Fehler während der Kommandogenerator nach dem Starten auf eine eingehende Verbindung wartet.

Error during waiting for reconnection: 10004

Während, nach einem Verbindungsabbruch, auf eine eingehende Verbindung gewartet wird, ist ein Fehler aufgetreten, z. B. wenn währenddessen der Port gewechselt worden ist.

Funktionssimulator:

Could not connect

Error during connecting to the server: 10061

Der Funktionssimulator konnte sich nicht mit dem höheren System verbinden.

8.3 Die Log-Datei

Error during reconnecting to the server: 10061

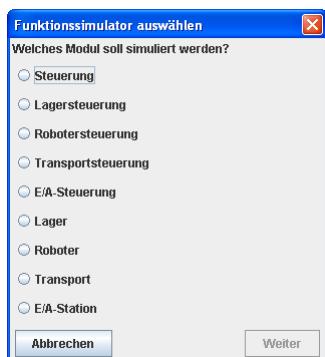
Der Funktionssimulator hat die Verbindung verloren und erfolglos sich probiert wieder zu verbinden.

9 Modul- und Buttonbeschreibung

9.1 Funktionssimulator

Im nun folgenden Abschnitt wird auf die einzelnen Dialoge bzw. Module des Funktionssimulator eingegangen. Es wird beschrieben, um welche Klasse es sich jeweils handelt und was die einzelnen Steuerungselemente für Bedeutungen haben. Dabei wird auch detailliert auf die Methoden eingegangen, welche hinter diesen stehen.

9.1.1 Auswahl des Funktionssimulators



Dialogfenster um den Funktionssimulator auszuwählen. Erscheint beim Starten des Programms und wenn man in der Menübar auf Datei -> „Funktionssimulator auswählen“ geht.

Abbrechen

Klasse: FSChoiceDialog

Methode: initButtons() – but_abort.addActionListener()

Beschreibung: Zum Abbrechen der Funktionssimulatormauswahl. Das Dialogfenster wird geschlossen ohne dass etwas geschieht.

Weiter

Klasse: FSChoiceDialog

Methode: initButtons() – but_next.addActionListener()

Beschreibung:

Wird freigeschalten, wenn ein Funktionssimulator markiert worden ist und veranlasst das der ConnectionOptionDialog angezeigt und der ausgewählte Funktionssimulator gestartet wird.

9.1 Funktionssimulator

Der Dialog wird geschlossen und danach wird zum Starten des ConnectionOptionDialog die Methode showDialog mit dem gewählten Funktionssimulator aufgerufen.

Zum Starten des Funktionssimulators wird ebenfalls der Text des ausgewählten Radio-Buttons geholt und damit die Initialisierungsmethode „initMainFrame()“ des MainFrames aufgerufen.

Verbindungsoptionen



Klasse: ConnectionsOptionsDialog

Methode: initButtons() – but_ok.addActionListener(new ButtonPressedHandler())

Beschreibung:

Es wird der Inhalt des Textfeldes, in dem der IP-Adresse steht, ausgelesen und überprüft ob es sich dabei um eine gültige IP-Adresse handelt. Wenn das der Fall ist, wird der Dialog geschlossen und die aufgelesene IP-Adresse und der Port an den ConnectionDialog mit der Methode showDialog übergeben.



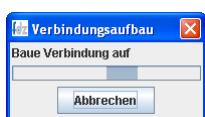
Klasse: ConnectionsOptionsDialog

Methode: initButtons() – but_abort.addActionListener()

Beschreibung:

Die vor dem Öffnen in zwei temporären Variablen gespeicherten Werte für IP und Port, werden wieder in die Textfelder geschrieben und danach das Dialogfenster geschlossen.

Verbindungsaufbau



Wenn dieser Dialog mit der Methode showDialog(String simulator, InetAddress address, short port) aufgerufen wird, startet der die ProgressBar und liest mittels der statischen Methode XmlModi.readXml() den Vector mit den Daten des im KGCoiceDialog ausgewählten Funktionssimulators aus. Diesen Simulator wird dann dem ConnectionPanel mittels der Methode setContollingSystem() gesetzt. Danach wird ein neuer FSMessageParser angelegt. Beim Anlegen werden den Parser unter anderem die Daten aus der Xml-Datei, die übergebene IPAdresse und der Port übergeben. Nachdem der FSMessageParser angelegt wurde, versucht dieser eine Verbindung zur übergebenen IP und auf dem Port aufzubauen. Wenn der Verbindungsaufbau erfolgreich war, schließt sich der Dialog von selbst.



Klasse: ConnectionsDialog

9.1 Funktionssimulator

Methode: initButton() – but_abort.addActionListener()

Beschreibung:

Bricht den Verbindungsauflauf ab indem er zuvor angelegte FSMessageParser wieder mittels der Methode closeConnection() wieder stoppt. Außerdem wird die Progressbar wieder angehalten und der Dialog geschlossen.

9.1.2 ModePanel



Im ModePanel kann der Modus des Funktionssimulators gewechselt werden. Man kann wählen zwischen Automatik und Test. Im Automatik-Modus werden eingehende Nachrichten selbstständig mit A001 und A002 quittiert. Im Test-Modus hat der Benutzer die Wahl was er auf eine eingehende Nachricht antworten will.

Automatik

Klasse: ModePanel

Methode: initComponents() - rb_automatic.addListener()

Beschreibung:

Wechselt den Funktionssimulator in den Automatik-Modus. Dafür wird die Funktion changeToTestModus(false) ausgeführt. Sie setzt dem FSMessageParser den neuen Modus.

Test

Klasse: ModePanel

Methode: initComponents() - rb_testMode.addListener()

Beschreibung:

Wechselt den Funktionssimulator in den Test-Modus. Dafür wird die Funktion changeToTestModus(true) ausgeführt. Sie setzt dem FSMessageParser den neuen Modus.

9.1 Funktionssimulator

9.1.3 ConnectionPanel



Das ConnectionPanel zeigt den aktuellen Status der Netzwerkverbindung an. Außerdem enthält das Panel ein Bild um den aktuellen FDZ-Teil zu veranschaulichen.



Klasse: ConnectionPanel

Methode: initButtons() - but_connect.addActionListener()

Beschreibung:

Dieser Button ermöglicht es eine Verbindung zu einem höheren FDZ-Teil aufzubauen. Dieser Button kann nur gedrückt werden, wenn keine Verbindung besteht. Nach Knopfdruck wird die Funktion connectToServer() aufgerufen. Sie startet den ConnectionOptionDialog mittels der Funktion showdDialog().



Klasse: ConnectionPanel

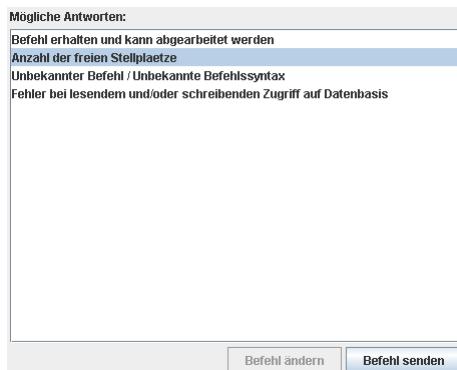
Methode: initButtons() - but_unlink.addActionListener()

Beschreibung:

Dieser Button ermöglicht es eine Verbindung zu einem höheren FDZ-Teil zu trennen. Dieser Button kann nur gedrückt werden, wenn eine Verbindung besteht. Nach Knopfdruck wird die Funktion disconnectFromServer() aufgerufen. Sie weist de FSMessageParser die Verbindung zu beenden mittels der Methode closeConnection(). Außerdem werden alle noch vorhandenen

Einträge im AnswerPanel gelöscht. Das geschieht mittels der Methode clearAnswerPanel().

9.1.4 AnswerPanel



Im AnswerPanel kann auf eingehende Nachrichten geantwortet werden. Dies ist allerdings nur möglich wenn sich der Funktionssimulator im Test-Modus befindet. Wenn ein Befehl eingegangen ist, fügt der FSMessageParser alle möglichen Antworten in das AnswerPanel mittels der Methode setData() ein. Der Benutzer kann dann einen Befehl auswählen, wenn nötig Eingabe-Werte einstellen und den erzeugten Befehl dann verschicken.

Befehl ändern oder

Klasse: AnswerPanel

Methode: initButtons() - but_change.addActionListener()

Beschreibung:

Mit diesem Button können für den ausgewählten Befehl alle nötigen Eingabe-Werte gesetzt werden. Durch Klick auf den Button wird die Methode createCommand(). Diese Methode schaut im Vector aus dem Xml nach, ob für den ausgewählten Befehl Input-Felder vorhanden sind. Wenn es Input-Felder gibt, wird der InputDialog gestartet damit der Benutzer die Input-Felder füllen kann. Sollte ein Befehl keine Eingabe-Felder besitzen so werden nur die Buttons neu gesetzt, damit der Befehl gesendet werden kann.

Befehl senden

Klasse: AnswerPanel

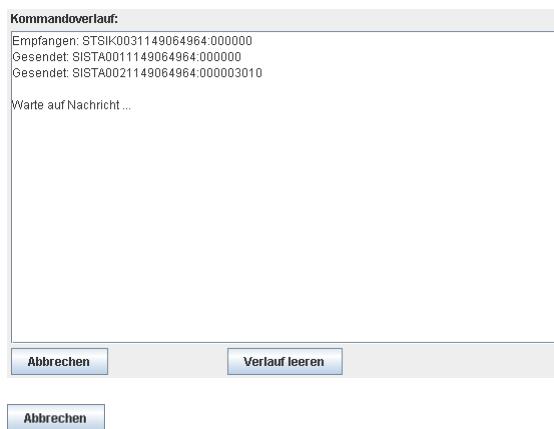
Methode: initButtons() - but_send.addActionListener()

Beschreibung:

Mit diesem Button kann ein vorher bearbeiteter Befehl versandt werden. Nach klicken des Buttons werden die Buttons neu gesetzt. D.h. der Befehl ändern Button wird wieder aktiviert und der Senden Button deaktiviert. Außerdem wird die Auswahl des Befehls wieder zurückgesetzt. Danach wird der Befehl mittels der Methode sendCommand(). Diese Methode weist den FSMessageParser an den Befehl mittels der Methode messageSend() zu verschicken.

9.1 Funktionssimulator

9.1.5 HistoryPanel



Klasse: HistoryPanel

Methode: initButton – but_abort.addActionListener()

Beschreibung:

Mit diesem Button kann eine gestartete Befehlsabfolge unterbrochen werden. Dafür werden die zwei Methoden setCommandToHistory() und clearAnswerPanel() ausgeführt. Die Funktion setCommandToHistory() setzt den Abgebrochen String in das HistoryPanel. Die Methode clearAnswerPanel() setzt das AnswerPanel zurück.

Verlauf leren

Klasse: HistoryPanel

Methode: initButton – but_clearList.addActionListener()

Beschreibung:

Löscht den Inhalt des History Panels und zeigt nur noch den momentan aktuellen Zustand des Funktionssimulators an.

Es wird die Methode „clearHistory()“ aufgerufen, wo die beiden Strings die den Inhalt des History Panels gespeichert haben gelöscht und die Text Area zurückgesetzt werden. Ist dies geschehen, wird der aktuelle Status des Funktionssimulators wieder in das HistoryPanel zurückgeschrieben.

9.1.6 Menüleiste

Menüleiste - Datei



Funktionssimulatorauswahl Strg-W

Klasse: FSMenuBar

9.1 Funktionssimulator

Methode: FSMenuBar() – mi_funktionssimulator_waehlen.addActionListener()

Beschreibung:

Öffnet das Dialogfenster, in dem es möglich ist zu einem anderen Funktionssimukator zu wechseln

Kommandoerlauf exportieren Strg-E

Klasse: FSMenuBar

Methode: FSMenuBar() – mi_kommandoerlauf_exportieren.addActionListener()

Beschreibung:

Die aktuelle Ansicht (Normal- / Detailansicht) wird in einer Datei gespeichert.

Es wird der Standarddialog von Java um Dateien abzuspeichern geöffnet, welchem dann der Pfad und der Dateiname entnommen werden. Mit diesen Daten ist es dann möglich, die aus dem History Panel geholten Daten über einen FileWriter in eine Datei zu schreiben.

Beenden Strg-B

Klasse: FSMenuBar

Methode: KGMenuBar() – mi_exit.addActionListener()

Beschreibung:

Dient zum Beenden des Programms.

Ruft die Methode „destroy()“ in der Klasse Main auf um das Programm zu beenden.

Menüleiste – Ansicht



Menüleiste – Ansicht - Normalansicht



Klasse: FSMenuBar

Methode: FSMenuBar() – mi_normal_kommandoauswahl.addActionListener()

FSMenuBar() – mi_normal_kommandoerlauf.addActionListener()

FSMenuBar() – mi_normal_alle.addActionListener()

Beschreibung:

Zeigt den Inhalt des ausgewählten Eintrages in der Normalansicht, d.h. die Nachrichten werden in einer leserlichen Form angezeigt.

9.1 Funktionssimulator

Das AnswerPanel hat die Daten von der Kommandoauswahlliste jeweils in zwei Vektoren gespeichert. Im einen sind die Einträge in einer leserlichen Form und im anderen Vektor in der Detailansicht. Genauso wird im History Panel beim Kommandoverlauf verfahren.

Soll nun die Kommandoauswahl und der Kommandoverlauf oder alle zwei in der Normalansicht angezeigt werden, wird jeweils nur im Answer bzw. History Panel die Methode zum Vertauschen der Vektoren aufgerufen und die Ansicht aktualisiert.

Menüleiste – Ansicht - Detailansicht



Klasse: FSMenuBar

Methode: FSMenuBar() – mi_detail_kommandoauswahl.addActionListener()

FSMenuBar() – mi_detail_kommandoüberlauf.addActionListener()

FSMenuBar() – mi_detail_alle.addActionListener()

Beschreibung:

Zeigt den Inhalt des ausgewählten Eintrages in der Detailansicht.

Das Answer Panel hat die Daten von der Kommandoauswahlliste jeweils in zwei Vektoren gespeichert. Im einen sind die Einträge in einer leserlichen Form und im anderen Vektor in der Detailansicht. Genauso wird im History Panel beim Kommandoverlauf verfahren.

Soll nun die Kommandoauswahl und der Kommandoverlauf oder alle zwei in der Detailansicht angezeigt werden, wird jeweils nur im Answer bzw. History Panel die Methode zum Vertauschen der Vektoren aufgerufen und die Ansicht aktualisiert.

Kommandoüberlauf leeren

Klasse: FSMenuBar

Methode: FSMenuBar() – mi_kommandoüberlauf_leeren.addActionListener()

Beschreibung:

Löscht den Inhalt des Kommandoverlaufs, so dass nur noch der aktuelle Status angezeigt wird.

Es wird dieselbe Methode „clearHistory()“ im HistoryPanel aufgerufen, wie wenn der Button „Liste leeren“ unter dem Kommandoverlauf gedrückt wird. Die Funktionsweise dieser Methode wird deshalb dort erklärt.

Menüleiste – Befehl



Da diese Befehle genauso fungieren, wie die Buttons im Answer Panel und auch dieselben Methoden verwenden, wird für die jeweilige Funktionsbeschreibung auf die dazugehörigen Buttons im Answer Panel verwiesen.

9.1 Funktionssimulator

Menüleiste – Modus



Da diese Befehle genauso fungieren, wie die RadioButtons im Mode Panel und auch dieselben Methoden verwenden, wird für die jeweilige Funktionsbeschreibung auf die dazugehörigen Buttons im Mode Panel verwiesen.

Menüleiste – Verbindung



Da diese Befehle genauso fungieren, wie die Buttons im Connection Panel und auch dieselben Methoden verwenden, wird für die jeweilige Funktionsbeschreibung auf die dazugehörigen Buttons im Connection Panel verwiesen.

Menüleiste – Optionen



Bei Klick auf Verbindungsoptionen wird der ConnectionOptionDialog gestartet.

Menüleiste - Hilfe



Hilfe [Strg-H](#)

Klasse FSMenuBar

Methode: FSMenuBar() – mi_help.addActionListener()

Beschreibung:

Startet die Hilfe. Dafür führt die in der MenuBar enthaltene Instanz der Klasse Help die Funktion showHelp() aus.

Info [Strg-I](#)

Klasse: FSMenuBar

Methode: FSMenuBar() – mi_about.addActionListener ()

Beschreibung:

Öffnet den Dialog für die Programminformationen (Version usw.).

Erzeugt einen MessageOptionsDialog mit den Informationen die angezeigt werden sollen und zeigt diesen an.

Menüleiste – Hilfe – Info

9.1 Funktionssimulator



Dialog mit Programminformationen (Version usw.)



Klasse: FSMenuBar

Methode: FSMenuBar() – mi_about.addActionListener ()

Beschreibung:

Schließt das Dialogfenster.

9.1.7 Input Dialog

Da der InputDialog die gleiche Funktion wie der InputDialog im Kommandogenerator hat, wird er hier nicht näher spezifiziert. ([Verweis zu Kommandogenerator : Input Dialog \(?? \)](#))

9.1.8 Smartiebelegungsdialog

Da der Smartiebelegungsdialog die gleiche Funktion wie der Smartiebelegungsdialog im Kommandogenerator hat, wird er hier nicht näher spezifiziert. ([Verweis zu Kommandogenerator : Schmartiebelegungsdialog\(?? \)](#))

9.1.9 JSAPInputDialog



Der JSAPInputDialog ist eine andere Version des InputDialogs. Der Unterschied zum InputDialog ist das der JSAPInputDialog einen Dialog mit 5 oder 6 Textfeldern anzeigt währenddessen der InputDialog dem Benutzer die Möglichkeit gibt eine Farbe und ein Textfeld zu füllen.



Klasse: JSAPInputDialog

Methode: initButton() – but_ok.addActionListener ()

Beschreibung:

9.2 Kommandogenerator

Erzeugt aus den Eingaben einen versandt fertigen Befehl und schließt das Dialogfenster. Sollte in einem Textfeld eine fehlerhafte Eingabe getätigt worden sein, erscheint ein Java Dialog der den Fehler meldet.

9.1.10 ErrorDialog

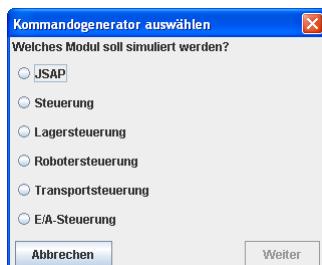
Da der ErrorDialog die gleiche Funktion wie der ErrorDialog im Kommandogenerator hat, wird er hier nicht näher spezifiziert. ([Verweis zu Kommandogenerator : ErrorDialog \(?? \)](#))

○ Test

9.2 Kommandogenerator

Im nun folgenden Abschnitt wird auf die einzelnen Dialoge bzw. Module des Kommandogenerators eingegangen. Es wird beschrieben, um welche Klasse es sich jeweils handelt und was die einzelnen Steuerungselemente für Bedeutungen haben. Dabei wird auch detailliert auf die Methoden eingegangen, welche hinter diesen stehen.

9.2.1 Auswahl des Kommandogenerators



Dialogfenster um den Kommandogenerator auszuwählen. Erscheint beim Starten des Programms und wenn man in der Menübar auf Datei -> „Kommandogenerator auswählen“ geht.

Abbrechen

Klasse: KGChoiceDialog

Methode: initButtons() – but_abort.addActionListener()

Beschreibung:

Zum Abbrechen der Kommandogeneratorauswahl.

Das Dialogfenster wird geschlossen ohne dass etwas geschieht.

Weiter

Klasse: KGChoiceDialog

Methode: initButtons() – but_next.addActionListener()

9.2 Kommandogenerator

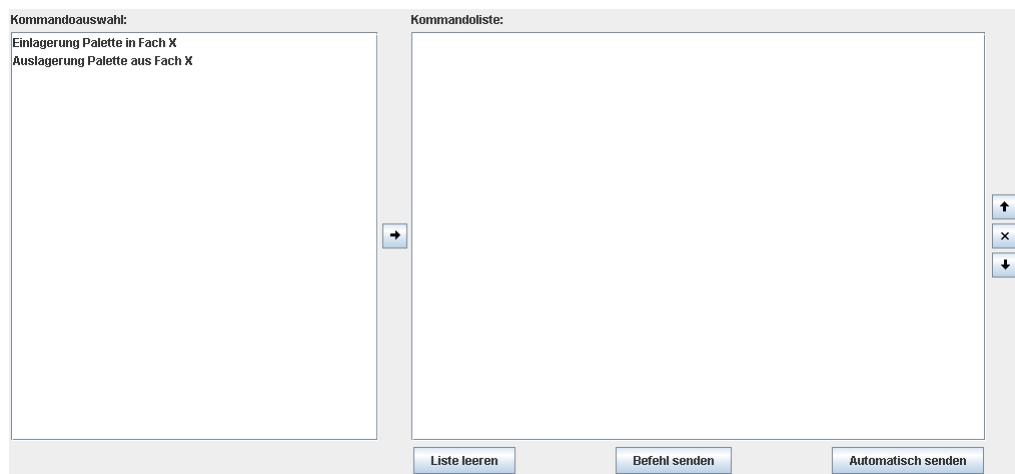
Beschreibung:

Wird freigeschalten, wenn ein Kommandogenerator markiert worden ist und veranlasst das der ausgewählte Kommandogenerator gestartet wird.

Es wird der Text des ausgewählten Radio-Buttons geholt und damit die Initialisierungsmethode „initMainFrame()“ des MainFrames aufgerufen. Danach wird das Dialogfenster geschlossen.

9.2.2 Command Panel

Command Panel - Kommandoauswahl



Das Command Panel ist Teil des MainFrames. Es listet in der Kommandoauswahlliste alle Kommandos auf, die vom aktuell gestarteten Kommandogenerator verschickt werden können. Diese können nach rechts zur Kommandoliste hinzugefügt werden. Dort ist es dann möglich sie in der Liste nach oben bzw. unten zu verschieben, sie einzeln herauszulöschen oder auch alle Einträge auf einmal zu entfernen. Des Weiteren kann man die Kommandos entweder manuell versenden oder die Einträge in der Kommandoliste automatisch nacheinander schicken lassen.



Klasse: CommandPanel

Methode: initListeners – but_pushToRight.addActionlistener() – pushToRight()

Beschreibung:

Fügt ein Kommando aus der Kommandoauswahlliste in die Kommandoliste ein.

Zunächst wird der Index des momentan selektierten Eintrages der Kommandoauswahlliste geholt. Im Falle dass dieser -1 ist, braucht nichts weiter gemacht zu werden, da dies bedeutet das nichts in der Liste ausgewählt worden ist. Ansonsten wird mit Hilfe des Indexes, das Kommando aus dem Vektor geholt, in dem alle für diesen Kommandogenerator möglichen Kommandos gespeichert sind.

Danach wird nachgeschaut, wie viele und was für Übergabeparameter dieses Kommando benötigt und der dafür notwendige Input Dialog geöffnet, damit diese Parameter vom Benutzer

9.2 Kommandogenerator

mit Werten belegt werden können. Sind keine Parameter nötig, wird die Methode „setMadeCommand()“ aufgerufen, die dann das Kommando der Kommandoliste hinzufügt.

Command Panel – Input Dialog

Soll ein Kommando aus der Kommandoauswahlliste zur Kommandoliste hinzugefügt werden, kann es sein das dieses Parameter benötigt, die mit gesendet werden sollen. Um diese eingeben zu können, gibt es verschiedene Dialogfenster, die dies ermöglichen. Jeder von ihnen ist schon im vornherein mit Default-Werten vorbelegt. Die verschiedenen Typen von Dialogfenstern zur Eingabe der Parameter sollen hier nun kurz erläutert werden.

Input Dialog - Typ I



Erscheint, wenn ein Befehl aus der Kommandoauswahlliste in die Kommandoliste eingefügt werden soll, der ein Argument benötigt das vom Typ Integer ist. Dies ist z. B. der Fall, wenn die Lagersteuerung beim Lager die Anzahl der freien Stellplätze abfragen möchte.

10

Beschreibung:

Eingabefeld für Ziffern, Buchstaben und Symbole. Einzige Einschränkung ist, es dürfen, je nach Befehl, nur eine bestimmte Anzahl an Zeichen sein. Im Normalfall wird hier eine Zahl eingegeben. Aber zu Testzwecken ist es auch möglich andere Zeichen einzugeben um das Zielsystem zu überprüfen.

OK

Klasse: InputDialog

Methode: initButtons() – but_ok.addActionListener()

Beschreibung:

Überprüft je Input Dialog die eingegebenen Werte und lässt bei einer fehlerhaften Eingabe eine Fehlermeldung erscheinen.

Zunächst wird nachgeschaut, um was für einen Typ von Input Dialog es sich handelt. Danach wird der Text aus dem Eingabefeld ausgelesen und auf seine Länge, also die Anzahl an Zeichen, überprüft. Sind es mehr Zeichen als benötigt, erscheint eine Fehlermeldung. Bei einer geringeren Anzahl, werden die noch fehlenden Stellen mit „0“ aufgefüllt (werden dem Eingegebenen vorangestellt). Ist die Anzahl der Zeichen (dann) richtig, wird das Eingegebene in die ausgewählte Nachricht eingefügt und dem Command Panel übergeben, damit sie zur Kommandoliste hinzugefügt werden kann.

Input Dialog – Typ II



9.2 Kommandogenerator

Dient zum Auswählen einer bestimmten Farbe von Smarties. Dies kann z. B. gebraucht werden, wenn die Steuerung von der Lagersteuerung die Anzahl der noch vorhandenen Smarties einer bestimmten Farbe wissen möchte.



Klasse: InputDialog

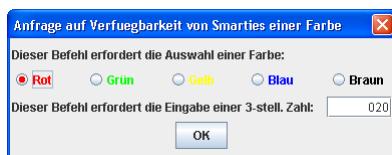
Methode: initButtons() – but_ok.addActionListener()

Beschreibung:

Schließt das Dialogfenster und übernimmt die Auswahl.

Fügt die getätigte Farbauswahl in das Kommando ein, das in Kommandoliste eingefügt werden soll und übergibt das nun erstellte Kommando dem Command Panel.

Input Dialog – Typ III



Ist die Mischung aus den Dialogen vom Typ I + II. Es ermöglicht die Auswahl einer Farbe und die Eingabe einer je nach Kommando verschiedenen Anzahl an Zeichen. Dieser Dialog wird z. B. gebraucht, wenn die Steuerung von der Lagersteuerung wissen will, ob so und soviel Smarties der ausgewählten Farbe noch vorhanden sind.



Beschreibung:

Eingabefeld für Ziffern, Buchstaben und Symbole. Einzige Einschränkung ist, es dürfen, je nach Befehl, nur eine bestimmte Anzahl an Zeichen sein. Im Normalfall wird hier eine Zahl eingegeben. Aber zu Testzwecken ist es auch möglich andere Zeichen einzugeben um das Zielsystem zu überprüfen.



Klasse: InputDialog

Methode: initButtons() – but_ok.addActionListener()

Beschreibung:

Überprüft die Eingabe und schließt das Dialogfenster.

Die Logik die hinter dem Button steckt, ist dieselbe, wie bei den Typen I + II, nur das sie gemeinsam zur Anwendung kommen. Deshalb findet sich die Methodenbeschreibung bei den zwei vorherigen Typen.

Command Panel - Schlittenpositionsdialog

9.2 Kommandogenerator



Mithilfe des Schlittenpositionsdialogs kann eine Schlittenposition ausgewählt werden. Mögliche Werte sind:

- „ro“
- „la“
- „ea“

Command Panel - Smartiebelegungsdialog

9.2 Kommandogenerator

C SmartieLayoutDialog	
▫ activeColor: String	
▫ answerPanel: AnswerPanel	
▫ bg: ButtonGroup	
▫ but_abort: JButton	
▫ but_ok: JButton	
▫ buttonPanel: JPanel	
▫ colors: Hashtable	
▫ columns: int	
▫ command: String	
▫ content: JPanel	
▫ description: String	
▫ fc: JFileChooser	
▫ images: Hashtable	
▫ F imgWidth: int	
▫ input: XmlInput	
▫ pFarben: JPanel	
▫ pFarbenWidth: int	
▫ palettenElemente: JLabel	
▫ rows: int	
S F serialVersionUID: long	
● F SmartieLayoutDialog(in owner: Frame, in modal: boolean, in answerPanel: AnswerPanel)	
● F SmartieLayoutDialog(in owner: Frame, in modal: boolean, in answerPanel: AnswerPanel, in layout: String)	
● actionPerformed(in e: ActionEvent)	
▫ buildDialog()	
▫ buildlayout(in anordnung: String)	
▫ genButtonPanel()	
▫ genColorPanel()	
● genImageLabel()	
▫ getPreview(): String	
▫ init(in layout: String)	
▫ initButtons()	
▫ loadImages()	
▫ posButtonPanel()	
▫ positionDialog(in width: int, in height: int)	
▫ setBackground()	
▫ setColor()	
● showDialog(in command: String, in description: String, in input: XmlInput)	

Wird benötigt, wenn für eine Nachricht als Übergabeparameter eine Palettenbelegung gebraucht wird. Diesen Dialog gibt es in zwei Varianten. Zum einen als Lagerpalette (13 Zeilen, 7 Spalten) und zum anderen als Produktpalette (9 Zeilen, 7 Spalten). Es besteht die Möglichkeit die verschiedenen Farben auszuwählen und damit die Palette zu bestücken, die Bestückung zu speichern und auch wieder zu laden. Beim erscheinen des Dialoges ist immer schon eine Defaultbelegung vorhanden.

Ablösen

Klasse: SmartieLayoutDialog

Beschreibung:

Schließt das Dialogfenster ohne die gemachten Belegungen zu übernehmen.

9.2 Kommandogenerator

In der Methode wird überprüft, welcher Button gedrückt worden ist und dann dementsprechend darauf reagiert. Hier z. B. wird das Dialogfenster einfach geschlossen ohne die Belegung weiterzuverwerten.



Klasse: SmartieLayoutDialog

Beschreibung:

Übernimmt die getätigte Auswahl und schließt den Dialog.

Erstellt einen String der die Auswahl in der Form „rygbw-“ enthält. Dabei stehen die Buchstaben für die jeweilige Farbe und „-“, wenn an dieser Stelle keine Farbe ist. Dieser wird dann zusammen mit der Nachricht, die der Kommandoliste hinzugefügt werden soll, dem KGMessageParser gegeben, welcher den erstellten String in diese Nachricht einfügt. Das fertige Kommando bekommt dann das Command Panel wieder zurück, um es in die Kommandoliste einzufügen.



Klasse: SmartieLayoutDialog

Beschreibung:

Setzt alle Positionen der Palette, mit der aktuell ausgewählten Farbe.



Klasse: SmartieLayoutDialog

Beschreibung:

Lädt eine Belegung der Palette aus einer Datei.



Klasse: SmartieLayoutDialog

Beschreibung:

Speichert die Belegung der Palette in eine Datei. Command Panel - Kommandoliste



Klasse: CommandPanel

Methode: initListeners – but_pushHigher.addActionlistener() – pushCommandUp()

Beschreibung:

Verschiebt das in der Kommandoliste selektierte Element eine Position nach oben.

Als erstes wird überprüft, ob der Index, den die Kommandoliste zurückgibt, größer als 0 ist. Dadurch wird zum einen überprüft, ob überhaupt ein Eintrag in der Kommandoliste ausgewählt ist und das dies nicht der erste Eintrag in der Liste ist. Wenn der zurückgegebene Index größer als 0 ist, kann das momentan selektierte Kommando in eine temporäre Variable zwischengespeichert werden. Jetzt werden, in den Vektoren mit den Daten der Kommandoliste, jeweils die Einträge

9.2 Kommandogenerator

an der Stelle des Indexes gelöscht, wodurch alle nachfolgenden Einträge eine Stelle nach vorne rutschen. Danach wird das Kommando aus der temporären Variable an die Stelle Index-1 in die Vektoren eingefügt. Dadurch wird der momentan an dieser Stelle stehende Eintrag, sowie alle nachfolgenden Einträge, eine Position nach hinten verschoben. Nun wird noch die Selektierung in der Liste geändert und die Komponente aktualisiert.



Klasse: CommandPanel

Methode: initListeners – but_pushDown.addActionlistener() – pushCommandDown()

Beschreibung:

Verschiebt das in der Kommandoliste selektierte Element eine Position nach unten.

Als erstes wird überprüft, ob der von der Kommandoliste zurückgegebene Index positiv ist, dies bedeutet nämlich dass ein Element in der Liste selektiert ist. Außerdem darf es sich nicht um das letzte Element handeln, da dies nicht mehr nach hinten verschoben werden kann.

Jetzt kann das momentan selektierte Kommando in eine temporäre Variable zwischengespeichert werden. Danach werden, in den Vektoren mit den Daten der Liste, jeweils die Einträge an der Stelle des Indexes gelöscht, wodurch alle nachfolgenden Einträge eine Stelle nach vorne rutschen. Dann wird das Kommando aus der temporären Variable an die Stelle Index+1 in die Vektoren eingefügt. Dadurch wird der momentan an dieser Stelle stehende Eintrag, sowie alle nachfolgenden Einträge, eine Position nach hinten verschoben. Nun wird noch die Selektierung in der Liste geändert und die Komponente aktualisiert.



Klasse: CommandPanel

Methode: initListeners – but_delete.addActionlistener() – removeMadeCommand()

Beschreibung:

Löscht das markierte Kommando aus der Kommandoliste.

Zunächst wird überprüft, ob der von der Kommandoliste zurückgegebene Index positiv ist, dies bedeutet nämlich dass ein Element in der Liste selektiert ist, und dass die Liste überhaupt noch Kommandos enthält.

Danach wird einfach mit Hilfe des Indexes das Kommando aus den Vektoren mit den erstellten Kommandos gelöscht und die Liste aktualisiert.



Klasse: CommandPanel

Methode: initListeners – but_clearList.addActionlistener() – clearCommandList()

Beschreibung:

Löscht die Kommandoliste.

Die beiden Vektoren mit den erstellten Kommandos werden geleert und die Ansicht aktualisiert.

9.2 Kommandogenerator

Befehl senden

Klasse: CommandPanel

Methode: initListeners – but_send.addActionlistener() – sendCommand()

Beschreibung:

Sendet das in der Kommandoliste ausgewählte Kommando.

Zunächst wird über das History Panel geprüft, ob sich der Kommandogenerator in einem Zustand befindet, in dem es erlaubt ist ein Kommando zu senden. Ist dies der Fall, wird die Methode „sendCommand(int index)“ mit dem Index des momentan markierten Kommandos aufgerufen.

In dieser wird nachgeschaut, ob überhaupt ein Element in der Liste selektiert ist und die Liste nicht leer ist. Danach wird das zu sendende Kommando dem KGMessageParser übergeben und wenn dieser erfolgreich gesendet hat, es aus den Vektoren herausgelöscht. Falls ein Fehler während des Sendens auftreten sollte, erscheint eine Fehlermeldung.

Automatisch senden

Klasse: CommandPanel

Methode: initListeners – but_sendAll.addActionlistener() – sendAllCommands()

Beschreibung:

Sendet alle Kommandos die in der Kommandoliste stehen, bis diese leer ist oder ein Fehler auftritt.

Als erstes wird nachgeschaut, in welchem Modus sich der Kommandogenerator befindet. Denn im automatischen Modus dient der Button zum Abbrechen des Sendens und im manuellen zum Starten des automatischen Vorgangs. Ist die Kommandoliste nicht leer und ist der Kommandogenerator in einem Zustand in dem das Senden erlaubt ist, wird der Thread zum Senden von allen Kommandos aus der Kommandoliste gestartet und der Text des Buttons von „Automatisch senden“ in „Abbrechen“ geändert.

Abbrechen

Klasse: CommandPanel

Methode: initListeners – but_sendAll.addActionlistener() – sendAllCommands()

Beschreibung:

Unterbricht den Modus in dem die Kommandos automatisch gesendet werden.

Als erstes wird nachgeschaut, in welchem Modus sich der Kommandogenerator befindet. Denn im automatischen Modus dient der Button zum Abbrechen des Sendens und im manuellen zum Starten des automatischen Vorgangs. Danach wird das Flag, das im Thread bei jedem Durchlauf abgefragt wird, auf „false“ gesetzt, damit der Thread beim nächsten Durchgang stoppt wird und als letztes der Text auf dem Button wieder in „Automatisch senden“ umgeändert.

History Panel

9.2 Kommandogenerator



Klasse: HistoryPanel

Methode: initButton – but_abort.addActionListener()

Beschreibung:

Wird auf eine eingehende Nachricht gewartet, so kann mit diesem Button das Warten abgebrochen werden.

Zunächst wird nachgeschaut, ob man momentan auf eine Nachricht wartet und sich der Kommandogenerator nicht im automatischen Modus befindet. Werden beide Kriterien erfüllt, kann das Warten abgebrochen werden.

[Liste leeren](#)

Klasse: HistoryPanel

Methode: initButton – but_clearList.addActionListener()

Beschreibung:

Löscht den Inhalt des History Panels und zeigt nur noch den momentan aktuellen Zustand des Kommandogenerators an.

Es wird die Methode „clearHistory()“ aufgerufen, wo die beiden Strings die den Inhalt des History Panels gespeichert haben gelöscht und die Text Area zurückgesetzt werden. Ist dies geschehen, wird der aktuelle Status des Kommandogenerators wieder in das HistoryPanel zurückgeschrieben.

9.2.3 Menüleiste

Datei Ansicht Befehl Optionen Hilfe

Menüleiste - Datei

Datei	Ansicht	Befehl	Optionen
Kommandogeneratorauswahl	Strg-W		
Kommandoliste speichern	Strg-S		
Kommandoliste laden	Strg-L		
Kommandoerlauf exportieren	Strg-E		
Beenden	Strg-B		

Kommandogeneratorauswahl Strg-W

Klasse: KGMenuBar

9.2 Kommandogenerator

Methode: KGMenuBar() – mi_kommandogenerator_waehlen.addActionListener()

Beschreibung:

Öffnet das Dialogfenster, in dem es möglich ist zu einem anderen Kommandogenerator zu wechseln.

Überprüft, ob sich der Kommandogenerator momentan im automatischen Modus befindet. Ist dies nicht der Fall, wird über das MainFrame Kommandogeneratorauswahldialog geöffnet.

Kommandoliste speichern Strg-S

Klasse: KGMenuBar

Methode: KGMenuBar() – mi_kommandoliste_speichern.addActionListener()

Beschreibung:

Den Inhalt der Kommandoliste in eine Datei speichern.

Öffnet den Standarddialog von Java um Dateien abzuspeichern, welchem dann der Pfad und der Dateiname entnommen wird. Mit diesen Daten ist es dann möglich einen File- und ObjektOutputStream zu erstellen, an die dann die vom CommandPanel geholte Kommandoliste gegeben wird.

Kommandoliste laden Strg-L

Klasse: KGMenuBar

Methode: KGMenuBar() – mi_kommandoliste_laden.addActionListener()

Beschreibung:

Ermöglicht es eine zuvor in einer Datei abgespeicherte Kommandoliste einzulesen.

Öffnet den Standarddialog von Java um Dateien zu laden, welchem dann der Pfad und der Dateiname entnommen wird. Mit diesen Daten ist es dann möglich einen File- und ObjektInputStream zu erstellen, um die Informationen aus der Datei zu holen. Diese werden an das CommandPanel weitergegeben und dort dann in der Kommandoliste angezeigt.

Kommandoerlauf exportieren Strg-E

Klasse: KGMenuBar

Methode: KGMenuBar() – mi_kommandoerlauf_exportieren.addActionListener()

Beschreibung:

Die aktuelle Ansicht (Normal- / Detailansicht) wird in einer Datei gespeichert.

Es wird der Standarddialog von Java um Dateien abzuspeichern geöffnet, welchem dann der Pfad und der Dateiname entnommen wird. Mit diesen Daten ist es dann möglich, die aus dem History Panel geholten Daten über einen FileWriter in eine Datei zu schreiben.

Beenden Strg-B

Klasse: KGMenuBar

Methode: KGMenuBar() – mi_exit.addActionListener()

9.2 Kommandogenerator

Beschreibung:

Dient zum Beenden des Programms.

Ruft die Methode „destroy()“ in der Klasse Main auf um das Programm zu beenden.

Menüleiste – Ansicht



Menüleiste – Ansicht - Normalansicht



Klasse: KGMenuBar

Methode: KGMenuBar() – mi_normal_kommandoauswahl.addActionListener()

KGMenuBar() – mi_normal_kommandoliste.addActionListener()

KGMenuBar() – mi_normal_kommandooverlauf.addActionListener()

KGMenuBar() – mi_normal_alle.addActionListener()

Beschreibung:

Zeigt den Inhalt des ausgewählten Eintrages in der Normalansicht, d.h. die Nachrichten werden in einer leserlichen Form angezeigt.

Das Command Panel hat die Daten von der Kommandoliste und der Kommandoauswahlliste jeweils in zwei Vektoren gespeichert. Im einen sind die Einträge in einer leserlichen Form und im anderen Vektor in der Detailansicht. Genauso wird im History Panel beim Kommandooverlauf verfahren.

Soll nun die Kommandoliste, Kommandoauswahl, Kommandooverlauf oder alle drei in der Normalansicht angezeigt werden, wird jeweils nur im Command bzw. History Panel die Methode zum vertauschen der Vektoren aufgerufen und die Ansicht aktualisiert.

Menüleiste – Ansicht - Detailansicht



Klasse: KGMenuBar

Methode: KGMenuBar() – mi_detail_kommandoauswahl.addActionListener()

KGMenuBar() – mi_detail_kommandoliste.addActionListener()

KGMenuBar() – mi_detail_kommandooverlauf.addActionListener()

KGMenuBar() – mi_detail_alle.addActionListener()

9.2 Kommandogenerator

Beschreibung:

Zeigt den Inhalt des ausgewählten Eintrages in der Detailansicht.

Das Command Panel hat die Daten von der Kommandoliste und der Kommandoauswahlliste jeweils in zwei Vektoren gespeichert. Im einen sind die Einträge in einer leserlichen Form und im anderen Vektor in der Detailansicht. Genauso wird im History Panel beim Kommandooverlauf verfahren.

Soll nun die Kommandoliste, Kommandoauswahl, Komandooverlauf oder alle drei in der Detailansicht angezeigt werden, wird jeweils nur im Command bzw. History Panel die Methode zum vertauschen der Vektoren aufgerufen und die Ansicht aktualisiert.

Kommandoliste leeren Klasse: KGMenuBar

Methode: KGMenuBar() – mi_kommmandoliste_leeren.addActionListener()

Beschreibung:

Löscht alle erstellten Kommandos aus der Kommandoliste.

Hier wird genauso Verfahren, wie wenn man im CommandPanel auf den Button „Liste leeren“ klickt. Zunächst wird nachgeschaut ob sich der Kommandogenerator im automatischen Modus befindet. Ist dies nicht der Fall, wird die Methode „clearCommandList()“ von der Klasse CommandPanel aufgerufen um die Vektoren zu leeren, die für die Kommandoliste zuständig sind.

Komandooverlauf leeren

Klasse: KGMenuBar

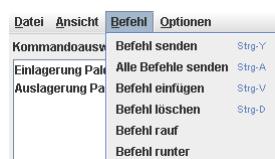
Methode: KGMenuBar() – mi_kommndooverlauf_leeren.addActionListener()

Beschreibung:

Löscht den Inhalt des Komandooverlaufs, so dass nur noch der aktuelle Status angezeigt wird.

Es wird dieselbe Methode „clearHistory()“ im HistoryPanel aufgerufen, wie wenn der Button „Liste leeren“ unter dem Komandooverlauf gedrückt wird. Die Funktionsweise dieser Methode wird deshalb dort erklärt.

Menüleiste – Befehl



Da diese Befehle genauso fungieren, wie die Buttons im Command Panel und auch dieselben Methoden verwenden, wird für die jeweilige Funktionsbeschreibung auf die dazugehörigen Buttons im Command Panel verwiesen.

Menüleiste – Optionen



Verbindungsoptionen Strg-O

9.2 Kommandogenerator

Klasse: KGMenuBar

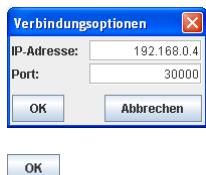
Methode: KGMenuBar() – mi_verbindungoptionen.addActionListener ()

Beschreibung:

Öffnet den Dialog für die Verbindungsoptionen.

Es wird über die Klasse Main das MainFrame geholt und mit diesem der Verbindungsoptionsdialog geöffnet.

Menüleiste – Optionen - Verbindungsoptionen



Klasse: ConnectionsOptionsDialog

Methode: initButtons() – but_ok.addActionListener()

Beschreibung:

Es wird der Inhalt des Textfeldes, in dem der Port steht, ausgelesen und mit dem beim Öffnen des Dialoges gespeicherten Wert verglichen.

Sind die beiden gleich, hat sich nichts verändert und das Dialogfenster kann geschlossen werden. Andernfalls, wird nachgeschaut, ob überhaupt etwas im Eingabefeld (Port) steht. Ist dies leer, dann erscheint eine Fehlermeldung. Andernfalls, wird der Wert im Feld zunächst in einer Property-Datei gespeichert und über das Command Panel veranlasst, dass der Port gewechselt wird. Ist dies alle geschehen, wird das Dialogfenster geschlossen.



Klasse: ConnectionsOptionsDialog

Methode: initButtons() – but_abort.addActionListener()

Beschreibung:

Der vor dem Öffnen in einer temporären Variable gespeicherte Wert, wird wieder in das Textfeld geschrieben und danach das Dialogfenster geschlossen.

Menüleiste - Hilfe



Hilfe Strg-H

Klasse: KGMenuBar

Methode: KGMenuBar() – mi_help.addActionListener ()

9.2 Kommandogenerator

Beschreibung:

Öffnet ein Fenster, das die Hilfe des Kommandogenerators enthält.

Es wird die Methode „showHelp()“ der Klasse Help aufgerufen, die das Fenster mit der Hilfe anzeigt.

Info Strg-I

Klasse: KGMenuBar

Methode: KGMenuBar() – mi_about.addActionListener ()

Beschreibung:

Öffnet den Dialog für die Programminformationen (Version usw.).

Erzeugt einen MessageOptionsDialog mit den Informationen die angezeigt werden sollen und zeigt diesen an.

Menüleiste – Hilfe – Info



Dialog mit Programminformationen (Version usw.)

OK

Klasse: KGMenuBar

Methode: KGMenuBar() – mi_about.addActionListener ()

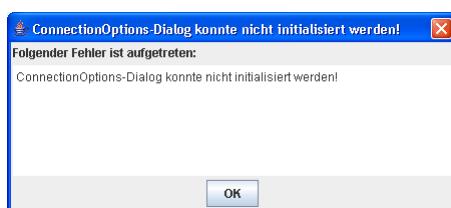
Beschreibung:

Schließt das Dialogfenster.

9.2.4 Error Dialog

Tritt ein Fehler während der Ausführung des Kommandogenerators auf, wird eine Fehlermeldung ausgegeben. Dabei wird zwischen zwei Typen von Fehlermeldungen unterschieden.

Error Dialog – Typ I



9.2 Kommandogenerator

Erscheint, wenn ein „normaler“ Fehler aufgetreten ist, der nicht so schwerwiegend ist. D. h. das Programm kann danach weiterhin ausgeführt werden. Ein Beispiel dafür wäre, wenn der Kommandooverlauf aus irgendeinem Grund nicht exportiert werden kann.

OK

Klasse: `ErrorDialog`

Methode: `actionPerformed()`

Beschreibung:

Schließt das Dialogfenster.

Error Dialog – Typ II



Wenn dieser Dialog erscheint, ist ein schwerwiegender Fehler aufgetreten, der die Funktionen des Kommandogenerators wesentlich einschränkt, z. B. wenn die XML-Datei nicht geladen werden konnte. Es ist dann zwar möglich das Programm weiterhin zu verwenden, jedoch nur beschränkt.

Ja

Klasse: `ErrorDialog`

Methode: `actionPerformed()`

Beschreibung:

Beendet das Programm.

Ruft die Methode „destroy“ der Klasse Main auf um das Programm zu beenden.

Nein

Klasse: `ErrorDialog`

Methode: `actionPerformed()`

Beschreibung:

Schließt das Dialogfenster.

Teil IV

Implementierung

10 Grundlegende Module

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

Im Folgenden wird das Verwenden der C-Kommunikations-Bibliothek im Detail beschrieben.

10.1.1 Funktionen

10.1.1.1 `ourselect`

Wrappet `select()`. `ourselect()` prüft, ob Daten auf den übergebenen Sockets vorhanden sind.

Prototyp:

- `int ourselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`

Parameter:

- `nfds`: Höchster Socket-Deskriptor + 1
- `*readfds`: Zeiger auf ein `fd_set` Objekt, in dem alle Sockets enthalten sind, die auf Lesbarkeit überprüft werden sollen.
- `*writefds`: wie `*readfds`, nur Überprüfung auf Schreibbarkeit
- `*exceptfds`: wie `*readfds`, nur Überprüfung auf alles andere
- `*timeout`: Enthält die Zeit in Sekunden und Millisekunden die `ourselect` die Sockets überprüfen soll

Rückgabewert:

- -1 bei Fehler
- 0 wenn keine Daten auf irgendeinem Socket sind
- $x > 0$: Anzahl der Sockets auf denen Daten liegen

Bemerkungen:

- keine

Fehlerbehandlung:

- Bei -1 wird der ErrorHandler aufgerufen.

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.1.2 initServerSocket

TCP/IP Server-Socket initialisieren

Prototyp:

- `int initServerSocket(unsigned short port);`

Parameter:

- `port`: Port-Nummer, auf der der Server erreichbar ist.

Rückgabewert:

- Socket-Descriptor bei Erfolg, NETWORK_ERROR bei Fehler

Bemerkungen:

- Die Funktion **initServerSocket** erzeugt einen neuen Server-Socket. Wurde der Server-Socket erfolgreich erzeugt, liefert die Funktion einen Socket-Descriptor zurück, im Fehlerfall wird NETWORK_ERROR zurückgeliefert. Der zurückgelieferte Socket-Descriptor muss an `awaitConnection` übergeben werden. Diese Funktion versetzt den Server in einen Wartezustand, in dem eingehende Verbindungen von Clients akzeptiert werden.

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit `setErrorHandler` gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor `initServerSocket` mit NETWORK_ERROR beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.1.3 awaitConnection

Bereits initialisierter TCP/IP Server-Socket wartet auf eingehende Clientverbindungen

Prototyp:

- int awaitConnection(int sockfd, int maxConnections);

Parameter:

- **sockfd**: Von initServerSocket zurückgelieferter Socket-Descriptor des Verbindungs-Sockets.
- **maxConnections**: Maximale Anzahl der gleichzeitigen Verbindungen. In der aktuellen Version der Kommunikations-Bibliothek muss für diesen Parameter immer 1 übergeben werden.

Rückgabewert:

- Socket-Descriptor bei Erfolg, NETWORK_ERROR bei Fehler

Bemerkungen:

- **awaitConnection** wartet auf eingehende Verbindungen von Clients. Die Funktion blockiert solange, bis sich ein Client verbindet. Als erster Parameter muss ein vorher mit initServerSocket erzeugter Verbindungs-Socket übergeben werden. Das zweite Argument ist auf 1 zu setzen. Der Rückgabewert ist ein Socket-Descriptor, der den Funktionen sendMessage und receiveMessage übergeben wird. Dieser Descriptor unterscheidet sich von dem an awaitConnection übergebenen Socket-Descriptor!

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit setErrorHandler gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor awaitConnection mit NETWORK_ERROR beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.1.4 openConnection

Initialisiert einen TCP/IP Client-Socket und baut eine Verbindung zu einem Server-Socket auf

Prototyp:

- int openConnection(const char* ip, unsigned short port);

Parameter:

- **ip:** Hostname oder IPv4 Zieladresse als nullterminierter C-String. Notation: "xxx.xxx.xxx.xxx". Führende Nullen können weggelassen werden.
- **port:** Portnummer des Servers

Rückgabewert:

- Socket-Descriptor bei Erfolg, NETWORK_ERROR bei Fehler

Bemerkungen:

- **openConnection** baut eine Clientverbindung zu einem unter **ip** und **port** erreichbaren Server-Socket auf. Der Rückgabewert ist ein Socket-Descriptor, der den Funktionen sendMessage und receiveMessage übergeben wird.

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit setErrorHandler gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor openConnection mit NETWORK_ERROR beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.1.5 sendMessage

Sendet eine Nachricht über einen geöffneten Socket

Prototyp:

- int sendMessage(int fd, SMessage* message);

Parameter:

- **fd**: Von awaitConnection oder openConnection zurückgelieferter Socket-Descriptor.
- **message**: Pointer auf SMessage Dateistruktur.

Rückgabewert:

- NETWORK_SUCCESS bei Erfolg, NETWORK_ERROR bei Fehler

Bemerkungen:

- **sendMessage** sendet den in der Datenstruktur SMessage enthaltenen String. Um einen String in die Datenstruktur zu schreiben, **muss** die Funktion initMessage verwendet werden.

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit setErrorHandler gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor sendMessage mit NETWORK_ERROR beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1.1.6 receiveMessage

Empfängt eine Nachricht auf einem geöffneten Socket

Prototyp:

- int receiveMessage(int fd, SMessage* message);

Parameter:

- **fd**: Von awaitConnection oder openConnection zurückgelieferter Socket-Descriptor.
- **message**: Pointer auf SMessage Dateistruktur.

Rückgabewert:

- NETWORK_SUCCESS bei Erfolg, NETWORK_ERROR bei Fehler

Bemerkungen:

- **receiveMessage** empfängt einen String und schreibt ihn in die Datenstruktur SMessage. Der Speicherplatz für diese Struktur muss durch den Anwender zur Verfügung gestellt werden. Diese Funktion blockiert, bis eine gültige Nachricht vollständig empfangen oder ein Fehler erkannt wurde.

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit setErrorHandler gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor initServerSocket mit NETWORK_ERROR beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1.1.7 initMessage

Initialisiert eine Nachricht, bevor sie mit sendMessage gesendet werden darf

Prototyp:

- void initMessage(SMessage* message, const char* msg_string);

Parameter:

- message : Pointer auf SMessage Datenstruktur.
- msg_string : Nullterminierter String, der die zu sendende Nachricht enthält.

Rückgabewert:

- keiner

Bemerkungen:

- **initMessage** Initialisiert eine Nachricht, bevor sie mit sendMessage gesendet werden darf. Die Nachricht wird in der als Parameter übergebenen Struktur vom Typ SMessage zurückgeliefert. Der Speicherplatz für diese Struktur muss durch den Anwender zur Verfügung gestellt werden. msg_string : enthält die nullterminierte Nachricht, wie in Abschnitt Die Funktion initialisiert den in der Struktur SMessage enthaltenen Payload-Pointer payload und die Länge der Nachricht len. payload verweist auf den Anfang der Nutzdaten der Nachricht.

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit setErrorHandler gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor initMessage mit NETWORK_ERROR beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1.1.8 initMessagePayloadLength

Initialisiert die Länge der Nutzdaten einer Nachricht, bevor sie mit sendMessage gesendet wird

Prototyp:

- void initMessagePayloadLength (SMessage* m);

Parameter:

- m : Pointer auf SMessage Datenstruktur.

Rückgabewert:

- keiner

Bemerkungen:

- **initMessagePayloadLength** initialisiert die Länge der Nutzdaten einer Nachricht, bevor sie mit sendMessage gesendet wird. Die Funktion wird intern von sendMessage aufgerufen, um die Korrektheit der im Nachrichten-Header eingetragenen Nutzdaten-Länge vor dem Senden sicherzustellen.

Die Funktion schreibt die Payload-Länge in den in der Struktur SMessage enthaltenen String message.

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.1.9 **closeSocket**

Schließt einen Socket

Prototyp:

- `int closeSocket (int fd);`

Parameter:

- `fd`: Socket-Deskriptor, der den Socket identifiziert, der geschlossen werden soll.

Rückgabewert:

- `NETWORK_SUCCESS` bei Erfolg, `NETWORK_ERROR` bei Fehler

Bemerkungen:

- **closeSocket** Schließt einen von `initServerSocket` oder `awaitConnection` zurückgelieferten Socket-Deskriptor. Zugriffe, die nach dem Aufruf dieser Funktion auf den Socket-Deskriptor erfolgen, führen zu undefiniertem Verhalten. **closeSocket** liefert `NETWORK_ERROR` zurück, wenn es sich bei dem übergebenen Wert um keinen gültigen Socket-Deskriptor handelt.

Im Allgemeinen müssen Clients mindestens einen Socket, Server mindestens zwei Sockets mit dieser Funktion schließen. Im Anschluss daran muss noch `cleanup` aufgerufen werden. Diese Funktion prüft unter anderem, ob für alle geöffneten Sockets `closeSocket` aufgerufen wurde.

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit `setErrorHandler` gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor `closeSocket` mit `NETWORK_ERROR` beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1.1.10 cleanup

Prüft auf noch nicht geschlossene Sockets und gibt Ressourcen frei

Prototyp:

- int cleanup(void);

Parameter:

- keine

Rückgabewert:

- NETWORK_SUCCESS bei Erfolg, NETWORK_ERROR bei Fehler

Bemerkungen:

- **cleanup** prüft auf noch nicht geschlossene Sockets und gibt Ressourcen frei. Bei jedem Aufruf von initServerSocket oder awaitConnection wird intern ein Zähler erhöht. **cleanup** liefert NETWORK_ERROR zurück, wenn dieser Zähler nicht 0 ist. Zusätzlich gibt die Funktion Ressourcen frei, die von der Kommunikations-Bibliothek verwendet wurden. Aus diesem Grund sollte **cleanup** immer aufgerufen werden, wenn das verwendende Programm die Funktionen der Kommunikations-Bibliothek nicht mehr benötigt.

Fehlerbehandlung:

- Wenn der ErrorHandler vorher mit setErrorHandler gesetzt wurde, wird dieser zusätzlich nach Auftreten des Fehlers aufgerufen, bevor cleanup mit NETWORK_ERROR beendet wird.

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1.2 Datenstrukturen

10.1.2.1 SMessage

Die Struktur **SMessage** enthält Informationen zu einer Nachricht, die gesendet werden soll bzw. empfangen wurde

Definition:

```
• typedef struct _SMessage
{
    char message [MAX_MESSAGELENGTH];
    unsigned short int len;
    char* payload;
} SMessage;
```

Elemente:

- **message**: Nullterminierter String, der die gesamte zu sendende Nachricht (inklusive Header) enthält. Der String kann maximal die Länge MAX_MESSAGELENGTH - 1 haben. Die maximale Nachrichtenlänge wurde hier spezifiziert.
- **len**: Speichert die tatsächliche Länge der Nachricht in Zeichen.
- **payload**: Zeiger auf den Anfang der Nutzdaten in der Nachricht. Ist die SMessage-Struktur korrekt initialisiert worden, zeigt **payload** auf das erste Zeichen in **message**, dass zu den Nutzdaten gehört.

Bemerkungen:

- Die Struktur **SMessage** enthält Informationen zu einer Nachricht, die gesendet werden soll bzw. empfangen wurde. In einem Objekt des Typs SMessage ist die Nachricht selbst als nullterminierter String, ihre Länge sowie ein Zeiger auf den Beginn der Nutzdaten gespeichert.
receiveMessage liefert eine korrekt initialisierte Struktur vom Typ SMessage zurück. sendMessage erwartet ein SMessage-Objekt, das zuvor korrekt mit initMessage initialisiert wurde.

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.3 ErrorHandler

Ein Mechanismus der Fehlererkennung bzw. -behandlung in der Kommunikations-Bibliothek ist der Error-Handler. Es handelt sich um eine Funktion, die vom Anwender gesetzt werden kann. Wurde ein Error-Handler gesetzt, wird die angegebene Funktion bei jedem Fehler, der bei der Anwendung der Funktionen der Kommunikations-Bibliothek auftreten kann, aufgerufen, bevor die Funktion, die den Fehler ausgelöst hat, mit Rückgabewert NETWORK_ERROR beendet wird. Die Kommunikations-Bibliothek übergibt dem Error-Handler bestimmte Argumente, anhand derer der Anwender mehr über den aufgetretenen Fehler in Erfahrung bringen kann.

Weitere Informationen:

- HandlerType
- setErrorHandler

Wichtiger Hinweis:

- Aufgrund des internen Aufbaus dürfen aus dem ErrorHandler heraus keine Aufrufe der Kommunikationsbibliothek erfolgen. Der ErrorHandler dient lediglich zur Lokalisierung des Fehlers. Eine geeignete Fehlerbehandlungsstrategie ist in Kapitel ?? beschrieben

10.1.3.1 HandlerType

Definiert den Prototypen einer Fehlerbehandlungs-Routine

Definition:

- `typedef int (*HandlerType)(int connID, int err_no,
const char* msg, const char* err_no_msg);`

Parameter:

- `connID`: Identifiziert die Verbindung, auf der der Fehler auftrat, indem hier der Socket-Deskriptor übergeben wird. Trat der Fehler bei keiner bestimmten Verbindung auf, beispielsweise in initServerSocket, bevor diese Funktion intern einen Socket erzeugt hat, so wird für `connID` -1 übergeben.
Trat ein Fehler bei cleanup auf, wird für diesen Parameter `NETWORK_ERROR_CLEANUP` übergeben.
- `err_no`: Liefert eine interne System-Fehlernummer zurück. Da intern POSIX verwendet wird, entsprechen die übergebenen Werte den ERRNO-Fehlerwerten von POSIX.
- `msg`: Nullterminierter String, der eine von der Kommunikations-Bibliothek erzeugt Beschreibung des Fehlers enthält. Zur Zeit enthält dieser String lediglich den Namen der Funktion, in der der Fehler auftrat.
- `err_no_msg`: System-Fehlernachricht, die zum angegebenen Fehlercode `err_no` gehört. Da intern POSIX verwendet wird, entsprechen die übergebenen Nachrichten den ERRNO-Fehlernachrichten von POSIX.

Wichtiger Hinweis:

- Aufgrund des internen Aufbaus dürfen aus dem ErrorHandler heraus keine Aufrufe der Kommunikationsbibliothek erfolgen. Der ErrorHandler dient lediglich zur Lokalisierung des Fehlers. Eine geeignete Fehlerbehandlungsstrategie ist in Kapitel ?? beschrieben

Bemerkungen:

- **HandlerType** definiert den Prototypen einer Fehlerbehandlungs-Routine. Die vom Anwender mit setErrorHandler gesetzte Error-Handler-Funktion muss einen bestimmten Aufbau haben, der von HandlerType beschrieben wird. Die Kommunikations-Bibliothek ruft diese Funktion im Falle eines Fehlers auf und übergibt der Funktion bestimmte Werte, anhand derer der Anwender mehr über den aufgetretenen Fehler in Erfahrung bringen kann.

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.3.2 setErrorHandler

Fehlerbehandlungsroutine setzen

Prototyp:

- void setErrorHandler(HandlerType handler);

Parameter:

- **handler**: Vom Benutzer definierte Fehlerbehandlungsroutine, wird im Fehlerfall aufgerufen. Da es sich bei diesem Parameter um einen Funktionszeiger handelt, kann man auch NULL bzw. 0 übergeben, was einen vorher gesetzten Error-Handler wieder löscht, d.h. dieser wird im Fehlerfall nicht mehr aufgerufen. Die Fehlerbehandlungsroutine muss den bereits oben erläuterten Prototypen haben.

Rückgabewert:

- keiner

Bemerkungen:

- **setErrorHandler** setzt eine neue, vom Benutzer definierte Fehlerbehandlungs-Routine, die immer dann aufgerufen wird, wenn bei der Verwendung der Funktionen der Kommunikations-Bibliothek ein Fehler auftritt. Der Error-Handler kann entsprechende Meldungen ausgeben oder Gegenmaßnahmen ergreifen. Durch Übergabe von 0 kann ein zuvor gesetzter Error-Handler deaktiviert werden, ebenso kann der Error-Handler zur Laufzeit mit dieser Funktion geändert werden.

Es ist jedoch meist sinnvoll, genau einmal einen Error-Handler zu setzen, d.h. setErrorHandler sollte als aller erste Funktion der Kommunikations-Bibliothek aufgerufen werden.

Wichtiger Hinweis:

- Aufgrund des internen Aufbaus dürfen aus dem ErrorHandler heraus keine Aufrufe der Kommunikationsbibliothek erfolgen. Der ErrorHandler dient lediglich zur Lokalisierung des Fehlers. Eine geeignete Fehlerbehandlungsstrategie ist in Kapitel ?? beschrieben

Beispiele:

- Beispielcode für Server
- Beispielcode für Client

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

10.1.4 Beispielcode Server

Listing 10.1: Beispielcode - Server

```
#include "fdzNetwork.h"

int Handler(int id, int err_no, const char* m, const char* err_m)
{
    printf("Error on connection %d: %s. Errno: %d, Errno Message: %s\n",
           id, m, err_no, err_m);

    return 0;
}

int main(void)
{
    int fd, sockfd, port = 30000;
    SMessage sendM, receiveM;

    // Error-Handler setzen ++++++
    setErrorHandler(Handler);

    // Server initialisieren ++++++
    if ((fd = initServerSocket(port)) == NETWORK_ERROR)
        return -1;

    // Auf eingehende Verbindungen warten
    ++++++
    if ((sockfd = awaitConnection(fd, 1)) == NETWORK_ERROR)
        return -1;

    // Nachricht initialisieren ++++++
    initMessage(&sendM, "SllaK001999999999:330003001");

    // Nachricht senden ++++++
    if (sendMessage(sockfd, &sendM) == NETWORK_ERROR)
        return -1;

    // Ack1 empfangen ++++++
    if (receiveMessage(sockfd, &receiveM) == NETWORK_ERROR)
        return -1;

    // Ack2 empfangen ++++++
    if (receiveMessage(sockfd, &receiveM) == NETWORK_ERROR)
        return -1;

    // Sockets schließen ++++++
    closeSocket(sockfd);
    closeSocket(fd);

    // Aufräumen ++++++
    cleanup();
```

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

```
    return 0;  
}
```

10.1.5 Beispielcode Client

Listing 10.2: Beispielcode - Client

```
#include "fdzNetwork.h"  
  
int Handler(int id, int err_no, const char* m, const char* err_m)  
{  
    printf("Error on connection %d: %s. Errno: %d, Errno Message: %s\n",  
        id, m, err_no, err_m);  
  
    return 0;  
}  
  
int main(void)  
{  
    int fd;  
    SMessage m;  
  
    // Error-Handler setzen ++++++++  
    setErrorHandler(Handler);  
  
    // Verbindung öffnen ++++++++  
    if ((fd = openConnection("127.0.0.1", 30000)) == NETWORK_ERROR)  
        return -1;  
  
    // Nachricht empfangen ++++++++  
    if (receiveMessage(fd, &m) == NETWORK_ERROR)  
        return -1;  
  
    // Ack1 initialisieren ++++++++  
    initMessage(&m, "laSlA0017658584858:010000");  
  
    // Ack1 senden ++++++++  
    if (sendMessage(fd, &m) == NETWORK_ERROR)  
        return -1;  
  
    // Ack2 initialisieren ++++++++  
    initMessage(&m, "laSlA0023451385891:010000");  
  
    // Ack2 senden ++++++++  
    if (sendMessage(fd, &m) == NETWORK_ERROR)  
        return -1;  
  
    // Socket schließen ++++++++  
    closeSocket(fd);  
  
    // Aufräumen ++++++++  
    cleanup();
```

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

```
    return 0;  
}
```

10.1.6 Beispielcode Fehlerbehandlung

Listing 10.3: Beispielcode -Fehlerbehandlung Server

```
#include "fdzNetwork.h"  
  
int g_Error = 0;  
  
// Error-Handler  
int TestErrorHandler(int connId, int err_no, const char* msg, const  
    char* err_msg)  
{  
    printf("Error on connection %d: %s. Errno: %d, Errno Message: %s\n",  
        connId, msg, err_no, err_msg);  
  
    g_Error = 1;  
  
    return 0;  
}  
  
// Verbindung initialisieren  
int initConnection(int port, int* fd)  
{  
    int sockfd;  
  
    // Server initialisieren ++++++++  
    if ((*fd = initServerSocket(port)) == NETWORK_ERROR)  
        return -1;  
  
    // Auf eingehende Verbindungen warten  
    // ++++++++  
    if ((sockfd = awaitConnection(*fd, 1)) == NETWORK_ERROR)  
        return -1;  
  
    return sockfd;  
}  
  
// Komplette Protokoll-Nachricht versenden und ACKs empfangen  
int doMessage(int sockfd, char* Message)  
{  
    SMessage sendM, receiveM;  
  
    // Nachricht initialisieren ++++++++  
    initMessage(&sendM, Message);  
  
    // Nachricht senden ++++++++  
    if (sendMessage(sockfd, &sendM) == NETWORK_ERROR)  
        return -1;  
  
    // Ack1 empfangen ++++++++
```

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

```
if (receiveMessage(sockfd, &receiveM) == NETWORK_ERROR)
    return -1;

// Ack2 empfangen ++++++
if (receiveMessage(sockfd, &receiveM) == NETWORK_ERROR)
    return -1;

return 0;
}

int main(void)
{
    int sockfd = -1, fd, port = 30000, ready = 0;

    char* Message = "SllaK001999999999:330003001";

    // Error-Handler setzen ++++++
    setErrorHandler(TestErrorHandler);

    // Schleife zum Versenden von Protokoll-Nachrichten
    while (!ready)
    {
        if ((sockfd = initConnection(port, &fd)) == -1)
            break;

        printf("Verbindung hergestellt!\n");

        // Solange die Verbindung fehlerfrei funktioniert, senden
        while (!g_Error)
        {
            // ...

            doMessage(sockfd, Message);

            // ...
        }

        g_Error = 0;

        // Sockets schließen ++++++
        closeSocket(sockfd);
        closeSocket(fd);

        cleanup();
    }

    return 0;
}
```

Listing 10.4: Beispielcode -Fehlerbehandlung Client

```
#include "fdzNetwork.h"
```

10.1 C-Kommunikations-Bibliothek - Funktionsbeschreibung

```
int TestErrorHandler(int connId, int err_no, const char* msg, const
                     char* err_msg)
{
    printf("Error on connection %d: %s. Errno: %d, Errno Message: %s\n",
           connId, msg, err_no, err_msg);

    return 0;
}

int main(void)
{
    int fd, ready = 0;

    setErrorHandler(TestErrorHandler);

    SMessage m;

    // Error-Handler setzen ++++++
    setErrorHandler(TestErrorHandler);

    // Verbindung öffnen ++++++
    if ((fd = openConnection("127.0.0.1", 30000)) == NETWORK_ERROR)
        return -1;

    printf("Client: Verbindung hergestellt!");

    while (!ready)
    {
        // Nachricht empfangen ++++++
        if (receiveMessage(fd, &m) == NETWORK_ERROR)
            break;

        // Ack1 initialisieren ++++++
        initMessage(&m, "laS1A0017658584858:010000");

        // Ack1 senden ++++++
        if (sendMessage(fd, &m) == NETWORK_ERROR)
            break;

        // Ack2 initialisieren ++++++
        initMessage(&m, "laS1A0023451385891:010000");

        // Ack2 senden ++++++
        if (sendMessage(fd, &m) == NETWORK_ERROR)
            break;
    }

    // Socket schließen ++++++
    closeSocket(fd);

    // Aufräumen ++++++
    cleanup();
}
```

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

```
    return 0;  
}
```

10.1.7 Bekannte Fehler

Beim Test der Kommunikationsbibliothek wurde folgendes Problem festgestellt:
Der Sender einer Nachricht kann nicht erkennen, dass beim Empfänger das Netzwerkkabel gezogen wurde, d.h. er bekommt dennoch die Bestätigung NETWORK_SUCCESS beim Aufruf von sendMessage. Dieser Fehler ist auf das verwendete TCP/IP Protokoll zurückzuführen, welches bereits bei erfolgreichem Beschreiben des Sendepuffers das Senden bestätigt.

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.1 Funktionsbeschreibung der C++-Klasse FDZNetworkJob

10.2.1.1 FDZNetworkJob

Erzeugen eines Objekts der C++-Klasse FDZNetworkJob.

Prototyp:

- FDZNetworkJob(std::string a_description , Property_t a_params);

Parameter:

- a_description: „Send“ für das Senden einer Nachricht;
„Receive“ für das Empfangen einer Nachricht;
„Select“ für das Überwachen der Serververbindungen;
- a_params: TODO!

Rückgabewert:

- keinen

Bemerkungen:

- Die C++-Klasse FDZNetworkJob ist von FDZJob abgeleitet und repräsentiert je nach Initialisierung einen Job für das Senden, Empfangen einer Nachricht oder das Überwachen der Serververbindungen.

Fehlerbehandlung:

- Keine.

10.2.1.2 operator()

Überladener Klammer-Operator der Klasse FDZNetworkJob.

Prototyp:

- `bool operator()();`

Parameter:

- keine

Rückgabewert:

- `true / false`

Bemerkungen:

- Mit dem Aufruf des Klammer-Operators wird das Senden, Empfangen oder Überwachen ausgelöst. Welches Ereignis stattfinden soll, wird mittels Übergabeparameter im Konstruktor festgelegt.

Fehlerbehandlung:

- keine

10.2.2 Funktionsbeschreibung der C++ Klasse FDZNetwork

10.2.2.1 C++-Klasse FDZNetwork

Erzeugen eines Objekts der C++-Klasse FDZNetwork.

Prototyp:

- FDZNetwork () ;

Parameter:

- keine

Rückgabewert:

- keinen

Bemerkungen:

- Die C++-Klasse FDZNetwork kann sowohl Server- als auch Client-Verbindungen verwalten. Dazu wurden Datenstrukturen für Server und Client kreiert, die die FDZNetwork-Klasse intern verwendet, um beliebig viele Server- bzw. Client-Verbindungen zu handeln. Diese werden im folgenden noch detailliert beschrieben.
- FDZNetwork wurde als Singleton implementiert, das heisst, der Konstruktor ist als `private` deklariert. Um an das aktuelle FDZNetwork-Objekt zu kommen, muss man die statische Methode `getInstance` aufrufen.

Fehlerbehandlung:

- Keine.

10.2.2.2 getHighestSocket

Ermittelt den höchsten Socket aller Verbindungen.

Prototyp:

- int getHighestSocket();

Parameter:

- keine

Rückgabewert:

- Socket-Descriptor

Bemerkungen:

- Diese Funktion wird nur intern aufgerufen. Sie ist notwendig um den ersten Parameter von select() zu bestimmen.
Nur für internen Gebrauch notwendig!

Fehlerbehandlung:

- keine

10.2.2.3 getReadableSocket

Ermittelt den Socket auf dem Daten liegen. Das übergebene fd_set* erhält die Funktion von select().

Prototyp:

- void getReadableSocket(fd_set*);

Parameter:

- fd_set* : .

Rückgabewert:

- keinen

Bemerkungen:

- Nur für internen Gebrauch notwendig!

Fehlerbehandlung:

- Keine

10.2.2.4 setTimeout

Setzt das Timeout für select().

Prototyp:

- void setTimeout(int Seconds, int uSeconds);

Parameter:

- Seconds : Sekunden
- uSeconds : Millisekunden

Rückgabewert:

- keinen

Bemerkungen:

- Nur sinnvoll in Kombination eines nicht-blockierenden select().

Fehlerbehandlung:

- Keine

10.2.2.5 setBlocking

Setzt ob select() blockierend oder nicht-blockierend ausgeführt wird.

Prototyp:

- void setBlocking(bool Blocking);

Parameter:

- Blocking : true / false

Rückgabewert:

- keinen

Bemerkungen:

- Setzt man select auf nicht-blockierend, so wartet select() bis zu einem Timeout auf Daten.

Fehlerbehandlung:

- Keine

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.2.6 connect

Auf Verbindungen der Clients warten und dann mit allen angegebenen Servern verbinden.

Prototyp:

- `bool connect(void);`

Parameter:

- keine

Rückgabewert:

- `true` bei Erfolg, `false` bei Fehler

Bemerkungen:

- Diese Funktion ist nur für die Steuerung gedacht. Da sich erst alle Subsysteme an der Steuerung anmelden müssen bevor diese sich mit JSAP verbindet, wurde das alles in eine Funktion eingebettet.
- `connect()` verbindet sich mit allen Systemen in der Reihenfolge, in der sie mit `addClient(...)` bzw `addServer(...)` definiert wurden!

Fehlerbehandlung:

- Keine

10.2.2.7 selectSubsystems

Überprüft ob eine Nachricht an den bekannten Client-Verbindungen wartet.

Prototyp:

- int selectSubsystems();

Parameter:

- keine

Rückgabewert:

- -1 bei Fehler
- 0 wenn keine Daten auf irgendeinen Socket sind
- $x > 0$: Anzahl der Sockets auf denen Daten anliegen

Bemerkungen:

- -

Fehlerbehandlung:

- Keine

10.2.2.8 selectServers

Überprüft ob eine Nachricht an den bekannten Server-Verbindungen wartet.

Prototyp:

- int selectServers();

Parameter:

- keine

Rückgabewert:

- -1 bei Fehler
- 0 wenn keine Daten auf irgendeinen Socket sind
- $x > 0$: Anzahl der Sockets auf denen Daten anliegen

Bemerkungen:

- -

Fehlerbehandlung:

- Keine

10.2.2.9 receive

Empfangen einer Nachricht.

Prototyp:

- `bool receive(SMessage* rMessage);`

Parameter:

- `rMessage`: Speicher für die empfangene Nachricht

Rückgabewert:

- `true` bei Erfolg, `false` bei Fehler

Bemerkungen:

- Empfangen einer Nachricht und speichern in einer SMessage Struktur. Dazu wird intern ausgelesen auf welcher Verbindung Daten vorhanden sind und speichert diese in der übergebenen SMessage Struktur.
Außerdem wurden Teile der Recovery-Funktionalität mit eingebettet.

Fehlerbehandlung:

- Keine

10.2.2.10 send

Senden einer Nachricht.

Prototyp:

- `bool send(SMessage* sMessage);`

Parameter:

- `sMessage`: Nachricht die geschickt werden soll

Rückgabewert:

- `true` bei Erfolg, `false` bei Fehler

Bemerkungen:

- Die Funktion erkennt selbstständig an wen die Nachricht gerichtet ist und schickt die Nachricht über die entsprechende Verbindung.
Außerdem wurden Teile der Recovery-Funktionalität mit eingebettet.

Fehlerbehandlung:

- Keine

10.2.2.11 reconnect

Baut eine abgebrochene Verbindung neu auf.

Prototyp:

- `bool reconnect ();`

Parameter:

- keine

Rückgabewert:

- `true` bei Erfolg, `false` bei Fehler

Bemerkungen:

- Die Funktion baut eine abgebrochene Verbindung neu auf, wenn `receive` 0 zurückgegeben hat.

Fehlerbehandlung:

- Keine

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.2.12 getInstance

Gibt eine Referenz des aktuellen FDZNetwork-Objekts zurück.

Prototyp:

- static FDZNetwork& getInstance ();

Parameter:

- keine

Rückgabewert:

- FDZNetwork-Referenz

Bemerkungen:

- Beim ersten Aufruf wird das FDZNetwork-Objekt neu angelegt. Die statische Funktion gibt eine Referenz auf das aktuelle FDZNetwork-Objekt zurück. Damit wird sichergestellt, dass immer nur genau ein FDZNetwork-Objekt existiert.

Fehlerbehandlung:

- Keine

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.3 Funktionsbeschreibung der C++-Klasse ServerCon

Die Klasse ServerCon stellt die Funktionen und Daten zur Verfügung, die der Server braucht, um sich mit einem Client in Verbindung setzen zu können.

10.2.3.1 ServerCon

Konstruktor der Klasse ServerCon.

Prototyp:

- `ServerCon();`

Parameter:

- keine

Rückgabewert:

- keiner

Bemerkungen:

- -

Fehlerbehandlung:

- keine

10.2.3.2 ~ServerCon()

Destruktor der Klasse ServerCon.

Prototyp:

- `~ServerCon();`

Parameter:

- keine

Rückgabewert:

- keiner

Bemerkungen:

- -

Fehlerbehandlung:

- keine

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.3.3 addClient

Hier werden die Daten des Clients in `std::vector<SERVERDATA>` Data gespeichert.

Prototyp:

- `bool addClient (int Port, std::string ID);`

Parameter:

- `Port` : Port auf den sich der Client verbinden wird.
- `ID` : ID des Clients (zwei Zeichen lange Zeichenkette, die genauso aufgebaut ist, wie der Sender/Empfänger im Protokollheader)

Rückgabewert:

- Liefert `true` zurück, wenn der Server-Socket erfolgreich initialisiert wurde.
- Liefert `false` zurück, ein Fehler aufgetreten ist.

Bemerkungen:

- -

Fehlerbehandlung:

- siehe Fehlerbehandlung für `initServerSocket ()` (Kapitel 12.1.1.1)

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.3.4 connect()

Der Server wartet auf Verbindungen der Clients in der Reihenfolge, in der die Clients dem Server mit addClient() bekannt gemacht wurden.

Prototyp:

- `bool connect (void);`

Parameter:

- keine

Rückgabewert:

- Liefert `true` zurück, wenn der Client sich erfolgreich verbunden hat.
- Liefert `false` zurück, ein Fehler aufgetreten ist.

Bemerkungen:

- -

Fehlerbehandlung:

- siehe Fehlerbehandlung für `awaitConnection()` (Kapitel 12.1.1.2)

10.2.3.5 connect(int, std::string)

Speichert die Daten des Clients in Data und wartet auf Verbindung eines Clients.

Prototyp:

- `bool connect (int Port, std::string ID);`

Parameter:

- `Port` : Port auf den sich der Client verbinden wird.
- `ID` : ID des Clients (zwei Zeichen lange Zeichenkette, die genauso aufgebaut ist, wie der Sender/Empfänger im Protokollheader)

Rückgabewert:

- liefert `true` zurück, wenn der Client sich erfolgreich verbunden hat.
- liefert `false` zurück, ein Fehler aufgetreten ist.

Bemerkungen:

- Hier ist nur die Verbindung mit einem Client möglich. Damit sich mehrere Clients verbinden können, braucht man die Funktion `connect()`, ausserdem braucht man die Funktion `addClient(...)`.

Fehlerbehandlung:

- siehe Fehlerbehandlung für `awaitConnection()` (Kapitel 12.1.1.2)

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.4 Funktionsbeschreibung der C++-Klasse ClientCon

Die Klasse ClientCon stellt die Funktionen und Daten zur Verfügung, die der Client braucht, um sich mit einem Server verbinden zu können.

10.2.4.1 ClientCon

Konstruktor der Klasse ClientCon.

Prototyp:

- ClientCon();

Parameter:

- keine

Rückgabewert:

- keiner

Bemerkungen:

- -

Fehlerbehandlung:

- keine

10.2.4.2 ~ClientCon()

Destruktor der Klasse ClientCon.

Prototyp:

- `~ClientCon();`

Parameter:

- keine

Rückgabewert:

- keiner

Bemerkungen:

- -

Fehlerbehandlung:

- keine

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.4.3 addServer

Hier werden die Daten des Servers - zu dem der Client sich verbinden möchte - in std::vector<CLIENTDATA> Data gespeichert.

Prototyp:

- void addServer (std::string ServerIP, int ServerPort,
std::string ServerID);

Parameter:

- ServerIP : IP-Adresse des Servers
- ServerPort : Port des Servers
- ServerID : ID des Servers (zwei Zeichen lange Zeichenkette, die genauso aufgebaut ist, wie der Sender/Empfänger im Protokollheader)

Rückgabewert:

- keiner

Bemerkungen:

- -

Fehlerbehandlung:

- -

10.2 C++ Kommunikations-Bibliothek - Funktionsbeschreibung

10.2.4.4 connect ()

Verbindet zu allen Servern, in der Reihenfolge, in der die Server durch addServer () zu Data hinzugefügt wurden.

Prototyp:

- `bool connect (void);`

Parameter:

- keine

Rückgabewert:

- Liefert `true` zurück, wenn die Verbindung zum Server hergestellt wurde.
- Liefert `false` zurück, wenn ein Fehler aufgetreten ist.

Bemerkungen:

- -

Fehlerbehandlung:

- siehe Fehlerbehandlung für openConnection () (Kapitel 12.1.1.3)

10.2.4.5 `connect(std::string, int, std::string)`

Fügt einen neuen Server Data hinzu und verbindet sich sofort mit ihm.

Prototyp:

- `bool connect (std::string ServerIP, int ServerPort,
std::string ServerID);`

Parameter:

- `ServerIP` : IP-Adresse des Servers
- `ServerPort` : Port des Servers
- `ServerID` : ID des Servers (zwei Zeichen lange Zeichenkette, die genauso aufgebaut ist, wie der Sender/Empfänger im Protokollheader)

Rückgabewert:

- Liefert `true` zurück, wenn die Verbindung zum Server hergestellt wurde.
- Liefert `false` zurück, wenn ein Fehler aufgetreten ist.

Bemerkungen:

- Hier kann der Client sich nur zu einem Server verbinden. Benötigt er mehrere Server, muss er die Funktion `connect ()` verwenden, ausserdem braucht man die Funktion `addServer(...)`.

Fehlerbehandlung:

- siehe Fehlerbehandlung für `openConnection ()` (Kapitel 12.1.1.3)

10.2.5 Datenstrukturen

10.2.5.1 SERVERDATA

Die Struktur SERVERDATA enthält alles, was der Server über den Client wissen muss.

Definition:

```
• typedef struct _SERVERDATA
{
    int Port;
    int InitSocket;
    int Socket;
    std::string ID;

} SERVERDATA;
```

Elemente:

- **Port**: Port des Clients
- **InitSocket**: Enthält einen Socket-Descriptor, wenn der Server-Socket erfolgreich initialisiert wurde und NETWORK_ERROR, wenn ein Fehler aufgetreten ist.
- **Socket**: Enthält einen Socket-Descriptor, wenn ein Client sich erfolgreich mit dem Server verbunden hat und NETWORK_ERROR, wenn ein Fehler aufgetreten ist.
- **ID**: ID des Client (zwei Zeichen lange Zeichenkette, die genauso aufgebaut ist, wie der Sender/Empfänger im Protokollheader): Hiermit erkennen die Funktionen **send()** und **receive()** automatisch an wen welche Nachricht gesendet werden muss bzw. wer eine Nachricht gesendet hat.

Bemerkungen:

- -

10.2.5.2 CLIENTDATA

Die Struktur `CLIENTDATA` enthält alles, was der Client über den Server wissen muss.

Definition:

```
• typedef struct _CLIENTDATA
  {
    std::string IP;
    int Port;
    unsigned int Socket;
    std::string ID;
  } CLIENTDATA;
```

Elemente:

- `IP`: IP-Adresse des Servers
- `Port`: Port des Servers
- `Socket`: Enthält einen Socket-Descriptor, wenn eine Verbindung zu einem Server-Socket aufgebaut werden konnte und einen NETWORK_ERROR, wenn ein Fehler aufgetreten ist.
- `ID`: ID des Servers (zwei Zeichen lange Zeichenkette, die genauso aufgebaut ist, wie der Sender/Empfänger im Protokollheader): Hiermit erkennen die Funktionen `send()` und `receive()` automatisch an wen welche Nachricht gesendet werden muss bzw. wer eine Nachricht gesendet hat.

Bemerkungen:

- -

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.1 Native Seite - Funktionsbeschreibung

Im Folgenden wird die C++ - seitige Implementierung der jniBridge im Detail beschrieben.

10.3.1.1 Typen

10.3.1.1.1 jniBridge.h::SJNICallbackData

Prototyp:

```
• //This struct wraps the Method and the Java Object that receive  
   the Callbacks and is used in the customData - Part of the  
   SCallbackData defined in FDZmessageHandler. Dont get confused  
   about these two types  
typedef struct tag_SJNICallbackData  
{  
    jmethodID methodID;  
    jobject callbackSink;  
} SJNICallbackData ;
```

Parameter:

- jmethodID methodID: Speichert die jmethodID, die Notwendig ist, um eine Java-Funktion aufzurufen
- jobject callbackSink: Speichert eine globale Referenz auf das Objekt, auf dem die in methodID hinterlegte Funktion ausgeführt werden soll.

Rückgabewert:

- SJNICallbackData Kein eigentlicher Rückgabewert; erzeugt eine Struktur mit dem Typ.

Bemerkungen:

- Vermittelt dieser Struktur werden die notwendigen Daten (Funktion und Objekt, auf dem die Funktion ausgeführt werden soll), die Notwendig sind, um den Aufruf der Java-Funktion aus dem C heraus möglich zu machen.
Wird im customData-Member von SCallbackData gespeichert und in callbackWrapper() verwendet.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.1.2 jniBridge.h::jniLastError

Prototyp:

```
• //Definition of the global struct holding the last error
typedef struct tag_jniLastError
{
    int sConnID;
    int sErr_no;
    const char* sMsg;
    const char* sErr_no_msg;
} jniLastError ;
```

Parameter:

- **int** sConnID: Siehe HandlerType
- **int** sErr_no: Siehe HandlerType
- **const char*** sMsg: Siehe HandlerType
- **const char*** sErr_no_msg: Siehe HandlerType

Rückgabewert:

- jniLastError Kein eigentlicher Rückgabewert; erzeugt eine Struktur mit dem Typ.

Bemerkungen:

- Vermittels dieser Struktur werden die Daten des letzten an jniErrorCallback() übergegebenen Fehler (siehe HandlerType für eine Beschreibung der Fehlerdatei) global gespeichert und in jniDoError() verwendet.
- **ACHTUNG!!!!!** Es gibt eine bekanntes Problem mit diesem Konstrukt, siehe jniErrorCallback()

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.2 Makros

10.3.1.2.1 jniBridge.cpp::JNI_REVISIONGEN

Prototyp:

- `#define JNI_REVISIONGEN(a,b)`

Parameter:

- a: Erwartet wird ein Integer als ReleaseNummer der DLL
- b: Erwartet wird ein Integer als VersionsNummer der DLL

Rückgabewert:

- `>>Expandiert<<`

Bemerkungen:

- Dieses Macro erzeugt die Werte, die von `Java_jniBridge_JNIBridge_getVersion()`, `Java_jniBridge_JNIBridge_getMajorVersion()` und `Java_jniBridge_JNIBridge_getMinorVersion()` zurückgegeben werden.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.2.2 jniBridge.cpp::PRINTDEBUG

Prototyp:

- ```
#define PRINTDEBUG(a);
//#define PRINTDEBUG(a) a ;
```

#### Parameter:

- a: Erwartet wird ein Ausdruck, der zu einem gültigen C-Funktionsaufruf evaluiert.

#### Rückgabewert:

- >>Expandiert<<

#### Bemerkungen:

- Mit diesem Macro sind etliche Debug-Ausgaben maskiert. Wenn die zweite Version des Macros ent-kommentiert und die erste Version kommentiert wird, so werden in der anschliessend erzeugten .dll alle Ausgaben aktiviert.
- **ACHTUNG** Wenn die .dll aus Java heraus verwendet wird, werden Die Ausgaben der Macros erst angezeigt, nachdem das Aufrufende Java-Programm beendet ist.

#### Fehlerbehandlung:

- Keins

#### Beispiele:

- ```
PRINTDEBUG(printf(`Hello World!\n`);)
```

10.3.1.3 Funktionen

10.3.1.3.1 jniBridge.cpp::JNI_OnLoad

Prototyp:

- `jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved)`

Parameter:

- `JavaVM *vm`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `void *reserved`: Von Java erzeugter Parameter, Siehe Java Api Doc

Rückgabewert:

- `jint`: Die Erwünschte Java-Runtime Version. Es wird die Konstante `JNI_VERSION_1_4` zurückgegeben.

Bemerkungen:

- Dies ist eine vorderfinierte Funktion, die von der Java - Runtime beim Laden der DLL aufgerufen wird. Hier wird der globale Error-Handler für alle jniBridge-Funktionen gesetzt, indem `jniErrorCallback()` an `setErrorHandler()` übergeben wird.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.3.2 jniBridge.cpp::callbackWrapper

Prototyp:

- **void** callbackWrapper(SCallbackData* data)

Parameter:

- SCallbackData* data: ToDo

Rückgabewert:

- **void**

Bemerkungen:

- Die Java-Callbacks können vom MessageHandler nicht direkt angegriffen werden, da die Calling-Konvention verletzt würde. Um das Problem zu lösen wird callbackWrapper() von Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerJNICallback() für ALLE Java-Callbacks registriert. Die Beziehung von Java-Object und Methode auf dem Objekt wird über den customData-Part des SCallbackData-Parameters hergestellt.
Vergleiche Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerJNICallback().

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.1.3.3 jniBridge.cpp::Java_jniBridge_fdzNetwork_JNINetwork_send

Prototyp:

- `JNIEXPORT void JNICALL Java_jniBridge_fdzNetwork_JNINetwork_send(JNIEnv * env, jobject jthis, jstring aMessage, jint aBufferID)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jobject jthis`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jstring aMessage`: ToDo
- `jint aBufferID`: ToDo

Rückgabewert:

- `void`

Bemerkungen:

- Diese Funktion extrahiert aus dem Java-String-Object `aMessage` eine Nachricht und erzeugt vermittels `initMessage()` daraus ein `SMessage`. Diese wird dann per `sendMessage()` verschickt.

Fehlerbehandlung:

- Wenn `sendMessage()` einen Fehler zurückgibt, wird `jniDoError()` aufgerufen.

Beispiele:

- Keins

10.3.1.3.4 jniBridge.cpp::Java_jniBridge_fdzNetwork_JNINetwork_recieve

Prototyp:

- `JNIEXPORT void JNICALL Java_jniBridge_fdzNetwork_JNINetwork_recieve(JNIEnv * env,
jobject jthis, jint aBufferID)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jobject jthis`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jint aBufferID`: ToDo

Rückgabewert:

- `jstring` Die Nachricht, die vom Netzwerk empfangen wurde.

Bemerkungen:

- Diese Funktion ruft `receiveMessage()` auf und erzeugt aus der zurückgegebenen `SMessage` einen Java-String, der ans Java zurückgegeben wird.

Fehlerbehandlung:

- Wenn `receiveMessage()` einen Fehler zurückgibt, wird `jniDoError()` aufgerufen.

Beispiele:

- Keins

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.1.3.5 jniBridge.cpp::Java_jniBridge_fdzNetwork_JNINetwork_initServerSocket

Prototyp:

- `JNIEXPORT void JNICALL Java_jniBridge_fdzNetwork_JNINetwork_recieve(JNIEnv * env,
 jobject jthis, jstring aDottedIPAdress, jshort aPort)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jobject jthis`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jstring aDottedIPAdress`: ToDo
- `jshort aPort`: ToDo

Rückgabewert:

- `jint`: Der erzeugte Socket-descriptor

Bemerkungen:

- Diese Funktion ruft `initServerSocket()` auf gibt den erzeugten Socket-Descriptor zurück

Fehlerbehandlung:

- Wenn `initServerSocket()` einen Fehler zurückgibt, wird `jniDoError()` aufgerufen.

Beispiele:

- Keins

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.1.3.6 jniBridge.cpp::Java_jniBridge_fdzNetwork_JNINetwork_awaitConnection

Prototyp:

- `JNIEXPORT void JNICALL Java_jniBridge_fdzNetwork_JNINetwork_awaitConnection(JNIEnv *`
`env, jobject jthis, jint aServerSocket)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jobject jthis`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jint aServerSocket`: ToDo

Rückgabewert:

- `void`

Bemerkungen:

- Diese Funktion ruft `awaitConnection()` auf.

Fehlerbehandlung:

- Wenn `awaitConnection()` einen Fehler zurückgibt, wird `jniDoError()` aufgerufen.

Beispiele:

- Keins

10.3.1.3.7 jniBridge.cpp::Java_jniBridge_fdzNetwork_JNINetwork_openConnection

Prototyp:

```
• JNICALL jint JNICALL  
    Java_jniBridge_fdzNetwork_JNINetwork_openConnection(JNIEnv *  
        env, jobject jthis, jstring aDottedIPAdress, jshort aPort)
```

Parameter:

- JNIEnv * env: Von Java erzeugter Parameter, Siehe Java Api Doc
- jobject jthis: Von Java erzeugter Parameter, Siehe Java Api Doc
- jstring aDottedIPAdress: ToDo
- jshort aPort: ToDo

Rückgabewert:

- jint

Bemerkungen:

- Diese Funktion erzeugt aus dem Java-String-Objekt jstring aDottedIPAdress eine IP-Adresse in **char []**-Form und ruft damit und mit dem aPort openConnection() auf.

Fehlerbehandlung:

- Wenn openConnection() einen Fehler zurückgibt, wird jniDoError() aufgerufen.

Beispiele:

- Keins

10.3.1.3.8 jniBridge.cpp::Java_jniBridge_fdzNetwork_JNINetwork_closeSocket

Prototyp:

- `JNIEXPORT void JNICALL Java_jniBridge_fdzNetwork_JNINetwork_closeSocket (JNIEnv *env, jobject jthis, jint aSocket)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jobject jthis`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jint aSocket`: ToDo

Rückgabewert:

- `void`

Bemerkungen:

- Diese Funktion ruft `closeSocket()` auf.

Fehlerbehandlung:

- Wenn `closeSocket()` einen Fehler zurückgibt, wird `jniDoError()` aufgerufen.

Beispiele:

- Keins

10.3.1.3.9 jniBridge.cpp::Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerCallback

Prototyp:

```
• JNIEXPORT void JNICALL Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerCallback  
    (JNIEnv *env, jclass ajclass, jobject aJNIMessageCallback)
```

Parameter:

- **JNIEnv * env**: Von Java erzeugter Parameter, Siehe Java Api Doc
- **jclass ajclass**: Von Java erzeugter Parameter, Siehe Java Api Doc
- **jobject aJNIMessageCallback**: Ein **JNIMessageCallback**, der in ein **SCallbackData** eingepackt und registriert werden soll.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion erzeugt ein **SCallbackData**-Objekt, und belegt das **message**-Member mit dem aus **JNIMessageCallback::getRegisteredMessage()** zurückgegeben String.
- Dann wird ein **SJNICallbackData**-Objekt erzeugt. Dessen **callbackSink**-member wird mit einer neu erzeugten globalen Java Referenz belegt (siehe Java Api Doc, **NewGlobalRef()**), um sicherzustellen, dass das registrierte **JNIMessageCallback**-Objekt nicht von Garbage Collector abgeräumt werden kann. Der **methodID**-Member wird mit der **jmethodID** der **JNIMessageCallback::messageRecieved()**-Funktion des registrierten Objectes vorbelegt. Diese wird in **callbackWrapper()** benötigt, um die entsprechende Java-Funktion aufzurufen.
- Schliesslich wird **callbackWrapper()** mit dem **SCallbackData**-Objekt als Callback bei **registerCallback()** registriert.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.3.10 `jniBridge.cpp::Java_jniBridge_fdzMessageHandler_JNIMessageHandler_unregisterCallback()`

Prototyp:

```
• JNIEXPORT void JNICALL Java_jniBridge_fdzMessageHandler_JNIMessageHandler_unregisterCallback(  
    JNIEnv *env, jclass ajclass, jobject aJNIMessageCallback)
```

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jclass ajclass`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jobject aJNIMessageCallback`: Ein `JNIMessageCallback`, der in ein `SCallbackData` eingepackt und registriert werden soll.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion geht den umgekehrten Weg, wie `Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerCallback()`.
- Diese Funktion erzeugt ein `SCallbackData`-Objekt, und belegt das `message`-Member mit dem aus `JNIMessageCallback::getRegisteredMessage()` zurückgegeben String.
- Dann wird ein `SJNICallbackData`-Objekt erzeugt. Dessen `callbackSink`-Member wird mit einer neu erzeugten globalen Java Referenz belegt (siehe Java Api Doc, `NewGlobalRef()`), um sicherzustellen, dass das registrierte `JNIMessageCallback`-Objekt nicht von Garbage Collector abgeräumt werden kann. Der `methodID`-Member wird mit der `jmethodID` der `JNIMessageCallback::messageRecieved()`-Funktion des registrierten Objectes vorbelegt. Diese wird in `callbackWrapper()` benötigt, um die entsprechende Java-Funktion aufzurufen.
- Schliesslich wird `callbackWrapper()` mit dem `SCallbackData`-Objekt als Callback bei `unregisterCallback()` unregistriert und die Globale Referenz wieder freigegeben.
- **ACHTUNG!!!!!!** Da in `Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerCallback()` die Referenz auf die erzeugte globale Referenz komplett in die Hoheit des `messageHandler` moduls übergeben wird, wird auf diesem Weg zwar der Callback tatsächlich deregistriert, aber die in [...] `registerCallback` erzeugte Globale Referenz nicht freigegeben. Der Effekt nennt sich „Pinning“ und führt zu einem Memory-Leak in Java. Die Lösung ist, das erzeugte `SCallbackData`-Objekt in den Callbacks zu suchen und aus DESSEN `callbackSink` die ursprüngliche Globale referenz zu extrahieren und freizugeben.

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

Die Logik dieser Funktion ist wäre identisch mit `parseMessage()`; nur darf Sie den Callback nicht ausführen, sondern ein Array mit `sMessageCallback` zurückgeben, die mit der Nachricht übereinstimmen.

Im Moment ist so eine Funktion jedoch noch nicht erstellt; das Problem ist aber minimal, da zur Zeit noch kein deregistrieren von Callbacks aus dem Java heraus stattfindet.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.1.3.11 jniBridge.cpp::Java_jniBridge_fdzMessageHandler_JNIMessageHandler_dispatch

Prototyp:

```
• JNIEXPORT jint JNICALL  
Java_jniBridge_fdzMessageHandler_JNIMessageHandler_dispatchJNIMessage  
(JNIEnv *env, jclass ajclass, jstring aMessage)
```

Parameter:

- JNIEnv * env: Von Java erzeugter Parameter, Siehe Java Api Doc
- jclass ajclass: Von Java erzeugter Parameter, Siehe Java Api Doc
- jstring aMessage: Ein Nachricht, die geparsed werden soll.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion packt aus dem Java-String Parameter „aMessage“ den darunterliegenden C-String aus und ruft damit `parseMessage()` auf. Der Rückgabewert von `parseMessage()` wird ins Java weitergereicht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.3.12 jniBridge.cpp::Java_jniBridge_JNIBridge_getVersion

Prototyp:

- `JNIEXPORT jstring JNICALL Java_jniBridge_JNIBridge_getVersion(JNIEnv * env, jclass ajclass)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jclass ajclass`: Von Java erzeugter Parameter, Siehe Java Api Doc

Rückgabewert:

- `jstring`: Ein Java-String-Präsentation der aktuellen DLL-Version. Der String hat das Format Rev_ ReleaseNummer _ VersionsNummer

Bemerkungen:

- Der String wird durch das Macro `JNI_REVISIONGEN()` erzeugt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.1.3.13 jniBridge.cpp::Java_jniBridge_JNIBridge_getMajorVersion

Prototyp:

- `JNIEXPORT jint JNICALL Java_jniBridge_JNIBridge_getMajorVersion (JNIEnv * env, jclass ajclass)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jclass ajclass`: Von Java erzeugter Parameter, Siehe Java Api Doc

Rückgabewert:

- `jint`: Die ReleaseNummer aktuellen DLL-Version.

Bemerkungen:

- Der Wert wird durch den ersten Parameter des Macro `JNI_REVISIONGEN()` erzeugt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.1.3.14 jniBridge.cpp::Java_jniBridge_JNIBridge_getMinorVersion

Prototyp:

- `JNIEXPORT jint JNICALL Java_jniBridge_JNIBridge_getMinorVersion (JNIEnv * env, jclass ajclass)`

Parameter:

- `JNIEnv * env`: Von Java erzeugter Parameter, Siehe Java Api Doc
- `jobject jthis`: Von Java erzeugter Parameter, Siehe Java Api Doc

Rückgabewert:

- `jint`: Die VersionsNummer aktuellen DLL-Version.

Bemerkungen:

- Der Wert wird durch den zweiten Parameter des Macros `JNI_REVISIONGEN()` erzeugt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.3.15 jniBridge.cpp::throwException

Prototyp:

- `void throwException(JNIEnv *env, const char * exceptionName, const char *message)`

Parameter:

- `JNIEnv *env`: Der an die aufrufende Funktion übergebene Zeiger auf die Java Runtime.
Wird benötigt, um Java-funktionen aufzurufen.
- `const char * exceptionName`: ToDo
- `const char *message`: ToDo

Rückgabewert:

- `void`

Bemerkungen:

- Die Funktion ist deprecated und wird funktional durch `jniDoError()` ersetzt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.3.1.3.16 jniBridge.cpp::jniErrorCallback

Prototyp:

- `int jniErrorCallback(int aConnID, int aErr_no, const char* aMsg, const char* aErr_no_msg)`

Parameter:

- `int aConnID`: siehe HandlerType
- `int aErr_no`: siehe HandlerType
- `const char* aMsg`: siehe HandlerType
- `const char* aErr_no_msg`: siehe HandlerType

Rückgabewert:

- `int` Immer Null. Siehe HandlerType.

Bemerkungen:

- Diese Funktion wird als genereller Handler von `JNI_OnLoad()` als ErrorHandler in `setErrorHandler()` gesetzt.
Es werden lediglich die 4 Parameter in globalen Variablen gespeichert.
- **ACHTUNG!!!** Diese Art des Error-Handling ist nicht Thread-Safe. Bei einer Kollision würde das aber lediglich dazu führen, dass die per `JNINetworkException` zurückgegebenen Werte falsch sein können. Diese Einschränkung wurde jedoch in Kauf genommen, da Fehler in der Kommunikationsbibliothek Thread-Safe erkannt werden, das zurückgeben aber massive Plattformabhängigkeit in jniBridge erzeugt hätte.

Fehlerbehandlung:

- Keins; dies Funktion ist Helperfunktion zur Fehlerbehandlung.

Beispiele:

- Keins

10.3.1.3.17 jniBridge.cpp::jniDoError

Prototyp:

- **void** jniDoError(JNIEnv * env, jniLastError ajniLastError)

Parameter:

- JNIEnv *env: Der an die aufrufende Funktion übergebene Zeiger auf die Java Runtime.
Wird benötigt, um Java-funktionen aufzurufen.
- jniLastError ajniLastError: ToDo

Rückgabewert:

- **void**

Bemerkungen:

- In dieser Funktion wird aus den per jniErrorCallback() global gespeicherten Daten eine Java - JNINetworkException erzeugt. Sollte die JNINetworkException vom Java Classloader nicht geladen werden können, oder sollte die Exception nicht auf dem Stack abgelegt werden können, so wird eine entsprechende Meldung ausgegeben und das Programm mit der Java-Built-in-Funktion FatalError sofort und hart beendet.

Fehlerbehandlung:

- Bei Fehlern wird das Programm mit der Java-Built-in-funktion FatalError sofort und hart beendet (!), da diese Funktion eine Helperfunktion zur Fehlerbehandlung ist, und fehlgeschlagene Fehlerbehandlung nicht weiter behandelt wird.

Beispiele:

- Keins

10.3.2 Java Seite - Funktionsbeschreibung

Im Folgenden wird die Java - seitige Implementierung im Detail beschrieben.

10.3.2.1 Klassen

Die Funktionalität der jniBridge wird in folgenden Klassen abgebildet:

- **class** JNIBridge
Diese Klasse stellt die Möglichkeit zu Verfügung, eine dll bzw. ein shared object zu importieren.
- **interface** FDZSocket
wird implementiert von:
 - **abstract class** JNINetwork
davon Abgeleitet:
 - * **class** FDZClientSocket
 - * **class** FDZServerSocket
- **class** JNINetworkException
davon Abgeleitet:
 - **class** FDZNetworkException
- **class** EventListenerListEx

10.3.2.2 Methoden

10.3.2.2.1 `JNIBridge::JNIBridge`

Prototyp:

- `public abstract class JNIBridge`

Parameter:

- keinen Die Klasse hat nur einen statischen-Initialisierungblock.

Rückgabewert:

- keinen Die Klasse hat nur einen statischen-Initialisierungblock.

Bemerkungen:

- Dies ist ein statischer Initialisierungblock, der aufgerufen wird, sobald der Java Classloader das erste mal die Klasse lädt. Statisch deshalb, da das Initialisieren der dll VOR dem ersten verwenden der Klasse geschehen muss. Insbesondere, da die DLL auch Methoden für `JNINetwork` und andere bereitstellt.
- Der Block vergleicht ausserdem, ob die dll im Filesystem die notwendige Version aufweist, siehe Fehlerbehandlung. Diese Klasse wird lediglich als Basisklasse für die eigentlichen jniKlassen verwendet und importiert alle Funktionen aus der `jniBridge.dll`.

Fehlerbehandlung:

- Beim Initialisieren prüft dieser Block die Version der DLL (vergleiche das Erstellen der DLL-Revision in `JNI_REVISIONGEN()`). Sofern die DLL-Version geringer ist, als die in den static Membern `static final int recommendedMinorVersion` bzw. `static final int recommendedMajorVersion` hinterlegten, so wird eine `RuntimeException` mit entsprechender Beschreibung geworfen.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.2 **JNIBridge::getVersion**

Prototyp:

- **static native** String getVersion()

Parameter:

- **void**

Rückgabewert:

- String Der String, der mit JNI_REVISIONGEN() erzeugt wurde.

Bemerkungen:

- Diese Methode kann verwendet werden, um den aktuellen Versionsstand der dll als String auszugeben. Es wird lediglich Java_jniBridge_JNIBridge_getVersion() aufgerufen.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.3.2.2.3 `JNIBridge::getMajorVersion`

Prototyp:

- `static native int getMajorVersion()`

Parameter:

- `void`

Rückgabewert:

- `int` Die Major Revision, die mit `JNI_REVISIONEN()` erzeugt wurde.

Bemerkungen:

- Diese Methode kann verwendet werden, um den aktuellen Versionsstand der dll als `int` auszugeben. Es wird lediglich `Java_jniBridge_JNIBridge_getMajorVersion()` aufgerufen.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.3.2.2.4 `JNIBridge::getMinorVersion`

Prototyp:

- `static native int getMinorVersion()`

Parameter:

- `void`

Rückgabewert:

- `int` Die Minor Revision, die mit `JNI_REVISIONEN()` erzeugt wurde.

Bemerkungen:

- Diese Methode kann verwendet werden, um den aktuellen Versionsstand der dll als `int` auszugeben. Es wird lediglich `Java_jniBridge_JNIBridge_getMinorVersion()` aufgerufen.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.3.2.2.5 **JNIBridge::copyFileFromJar**

Prototyp:

- **private static void** copyFileFromJar(String fileName)

Parameter:

- String fileName: Name des Files, das extrahiert werden soll.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Methode kann verwendet werden, um Files aus dem aktuellen .jar auszupacken, wenn die Anwendung aus einem Jar-File gestartet wird.

Fehlerbehandlung:

- Es wird lediglich ein Stacktrace erzeugt und eine eventuelle Exception discarded. (Insbesondere FileNotFoundExceptions können auftreten.)

Beispiele:

- Keines

10.3.2.2.6 jniBridge::fdzNetwork.FDZSocket FDZSocket

Prototyp:

- `public interface FDZSocket`

Parameter:

- keinen Dies ist eine Interface.

Rückgabewert:

- keinen Dies ist eine Interface.

Bemerkungen:

- Dieses Interface ist für die eigentlichen jniNetwork - Klassen FDZServerSocket und FDZClientSocket definiert.

Fehlerbehandlung:

- Keines, dies ist ein Interface

Beispiele:

- Keines, dies ist ein Interface

10.3.2.2.7 jniBridge::send

Prototyp:

- `public void send(String aMessage) throws JNINetworkException`

Parameter:

- `String aMessage` Die zu sendende Nachricht.

Rückgabewert:

- `void`

Bemerkungen:

- Sendet eine Nachricht übers Netzwerk.

Fehlerbehandlung:

- Dies ist nur eine Interface-Methode

Beispiele:

- Keines, da dies ein Interface ist.

10.3.2.2.8 FDZSocket::recieve

Prototyp:

- **public** String recieve() **throws** JNINetworkException

Parameter:

- **void**

Rückgabewert:

- String: Die empfangene Nachricht.

Bemerkungen:

- Empfängt eine Nachricht vom Netzwerk.

Fehlerbehandlung:

- Dies ist nur eine Interface-Methode

Beispiele:

- Keines, da dies ein Interface ist.

10.3.2.2.9 FDZSocket::isConnected

Prototyp:

- `public boolean isConnected()`

Parameter:

- `void`

Rückgabewert:

- `boolean`: Gibt den Zustand des Sockets an. Siehe `JNINetworktriggerStateChange`

Bemerkungen:

- Gibt an, ob der Socket verbunden ist, oder nicht. Siehe `JNINetworktriggerStateChange`

Fehlerbehandlung:

- Dies ist nur eine Interface-Methode

Beispiele:

- Keines, da dies ein Interface ist.

10.3.2.2.10 **JNINetwork:::JNINetwork extends JNIBridge**

Prototyp:

- **protected** JNINetwork(**short** port)
- **protected** JNINetwork(InetAddress aNetworkAdress, **short** port)

Parameter:

- **short** port: Der Port, auf dem der Server hören soll bzw. zu dem der Client verbinden soll

Optional InetAddress aNetworkAdress: Die Adresse, auf der der Server hören soll, bzw. zu der der Client verbinden soll.

Rückgabewert:

- JNINetwork: Eine Instanz der Klasse

Bemerkungen:

- Diese beiden Konstruktoren belegen nur Membervariablen mit den Daten. Wird aNetworkAdress nicht belegt, wird die Adresse per InetAddress.getLocalHost() bestimmt.
Diese Klasse wird lediglich als Basisklasse für die eigentlichen jniKlassen verwendet und importiert die Funktionen aus der jniBridge.dll.

Fehlerbehandlung:

- Sollte InetAddress.getLocalHost() eine Exception werfen, wird ein Stacktrace erzeugt; die Exception wird allerdings discarded.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.11 JNINetwork::closeSocket

Prototyp:

- `public synchronized void closeSocket () throws FDZNetworkException`

Parameter:

- `void`

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode schliesst den zum aktuellen Objekt gehörenden Socket (egal ob Server oder Clientseitig), indem `Java_jniBridge_fdzNetwork_JNINetwork_closeSocket ()` aufgerufen wird.

Fehlerbehandlung:

- Sollte eine Exception aus `Java_jniBridge_fdzNetwork_JNINetwork_closeSocket ()` geworfen werden, so wird der Status des aktuellen Objekts auf „DISCONNECTED“ gesetzt, indem `JNINetwork::triggerStateChange()` aufgerufen wird. Dann wird eine `FDZNetworkException` aus der ursprünglichen Exception erzeugt und geworfen.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.12 **JNINetwork::closeServerSocket**

Prototyp:

- **public void** closeServerSocket () **throws** FDZNetworkException

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Diese Methode schliesst den zum aktuellen Objekt gehörenden ServerSocket, indem `Java_jniBridge_fdzNetwork_JNINetwork_closeSocket()` aufgerufen wird.
- Diese Funktion ist nicht synchronized, da es notwendig sein kann, einen ServerSocket zu schliessen, während ein anderer Thread in `JNINetwork::recieve()` steht.

Fehlerbehandlung:

- Sollte eine Exception aus `Java_jniBridge_fdzNetwork_JNINetwork_closeSocket()` geworfen werden, so wird der Status des aktuellen Objekts auf „DISCONNECTED“ gesetzt, indem `JNINetwork::triggerStateChange()` aufgerufen wird. Dann wird eine `FDZNetworkException` aus der ursprünglichen Exception erzeugt und geworfen.
- **Achtung!!!** Den Status auf „DISCONNECTED“ zu stellen ist eventuell etwas voreilig, da ein geschlossener ServerSocket nicht zwingend auch eine Abgebrochene Verbindung bedeutet. Da der Zustand aber vorkommen kann, wird der Status hier immer getriggert.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.13 **JNINetwork::initServerSocket**

Prototyp:

- **protected synchronized void** initServerSocket () **throws**
FDZNetworkException

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Diese Methode erzeugt einen ServerSocket, indem Java_jniBridge_fdzNetwork_JNINetwork_initServerSocket () aufgerufen wird.

Fehlerbehandlung:

- Sollte eine Exception aus Java_jniBridge_fdzNetwork_JNINetwork_initServerSocket () geworfen werden, wird eine FDZNetworkException aus der ursprünglichen Exception erzeugt und geworfen.
- Sollte der ServerSocket bereits erzeugt sein, wird eine RuntimeException geworfen. Dies hat den Grund, dass diese Art Exception nicht im throws-Block deklariert werden muss, und somit auch kein try-catch-Block deklariert werden muss. Diese Exception sollte überhaupt nicht vorkommen können. Da es während der Entwicklungsphase aber doch geschehen ist, wurde die Exception beibehalten.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.14 **JNINetwork::awaitConnection**

Prototyp:

- **protected synchronized void** awaitConnection() **throws**
FDZNetworkException

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Diese Methode wartet auf eine Clientverbindung zum aktuellen Objekt gehörenden ServerSocket, indem `Java_jniBridge_fdzNetwork_JNINetwork_awaitConnection()` aufgerufen wird.
- Wenn der Aufruf erfolgreich war, wird der Status des aktuellen Objects auf „CONNECTED“ gesetzt.

Fehlerbehandlung:

- Sollte eine Exception aus `Java_jniBridge_fdzNetwork_JNINetwork_awaitConnection()` geworfen werden, wird eine FDZNetworkException aus der ursprünglichen Exception erzeugt und geworfen.
- Sollte der ServerSocket noch verbunden sein, wird eine RuntimeException geworfen. Dies hat den Grund, dass diese Art Exception nicht im throws-Block deklariert werden muss, und somit auch kein try-catch-Block deklariert werden muss. Diese Exception sollte überhaupt nicht vorkommen können. Da es während der Entwicklungsphase aber doch geschehen ist, wurde die Exception beibehalten.
- Sollte der ServerSocket noch nicht initialisiert sein, wird eine RuntimeException geworfen. Diese ist ein Notnagel gegen das Problem im Konstruktor von FDZServerSocket.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.15 `JNINetwork::openConnection`

Prototyp:

- `protected synchronized void openConnection() throws JNINetworkException`

Parameter:

- `void`

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode erstellt eine Verbindung zum im Konstruktor angegebenen IP:Port Endpunkt, indem `Java_jniBridge_fdzNetwork_JNINetwork_openConnection()` aufgerufen wird.
- Wenn der Aufruf erfolgreich war, wird der Status des aktuellen Objects auf „CONNECTED“ gesetzt, indem `JNINetwork::triggerStateChange()` aufgerufen wird.

Fehlerbehandlung:

- Sollte eine Exception aus `Java_jniBridge_fdzNetwork_JNINetwork_openConnection()` geworfen werden, wird diese unverändert weiter geworfen, da das Handling dieser Exception in der Abgeleiteten Klasse erledigt wird.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.16 JNINetwork::recieve

Prototyp:

- `public String recieve() throws JNINetworkException`

Parameter:

- `void`

Rückgabewert:

- `String` Die empfangene Nachricht.

Bemerkungen:

- Liese Nachrichten vom Netzwerk, indem `Java_jniBridge_fdzNetwork_JNINetwork_recieve()` aufgerufen wird.

Fehlerbehandlung:

- Sollte eine Exception aus `Java_jniBridge_fdzNetwork_JNINetwork_recieve()` geworfen werden, so wird der Status des aktuellen Objekts auf „DISCONNECTED“ gesetzt, indem `JNINetwork::triggerStateChange()` aufgerufen wird. Die Exception wird anschliessend unverändert weiter geworfen, da das Handling dieser Exception in der Abgeleiteten Klasse erledigt wird.
- Sollte der Socket nicht verbunden sein, wird eine `RuntimeException` geworfen. Dies hat den Grund, dass diese Art Exception nicht im throws-Block deklariert werden muss, und somit auch kein try-catch-Block deklariert werden muss. Diese Exception sollte überhaupt nicht vorkommen können. Da es während der Entwicklungsphase aber doch geschehen ist, wurde die Exception beibehalten.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.17 **JNINetwork::send**

Prototyp:

- **public void** send(String aMessage) **throws** JNINetworkException

Parameter:

- String aMessage Die zu sendende Nachricht.

Rückgabewert:

- **void**

Bemerkungen:

- Sendet eine Nachricht übers Netzwerk, indem `Java_jniBridge_fdzNetwork_JNINetwork_send()` aufgerufen wird.

Fehlerbehandlung:

- Sollte eine Exception aus `Java_jniBridge_fdzNetwork_JNINetwork_send()` geworfen werden, so wird der Status des aktuellen Objekts auf „DISCONNECTED“ gesetzt, indem `JNINetwork::triggerStateChange()` aufgerufen wird. Die Exception wird anschliessend unverändert weiter geworfen, da das Handling dieser Exception in der Abgeleiteten Klasse erledigt wird.
- Sollte der Socket nicht verbunden sein, wird eine `RuntimeException` geworfen. Dies hat den Grund, dass diese Art Exception nicht im throws-Block deklariert werden muss, und somit auch kein try-catch-Block deklariert werden muss. Diese Exception sollte überhaupt nicht vorkommen können. Da es während der Entwicklungsphase aber doch geschehen ist, wurde die Exception beibehalten.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.18 JNINetwork::finalize

Prototyp:

- `public void finalize()`

Parameter:

- `void`

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode kann verwendet werden, um alle Sockets, die mit dem aktuellen Objekt assoziiert sind, komplett zu schliessen.
Sie ruft `JNINetwork::closeServerSocket()` und `JNINetwork::closeSocket()` auf.
- Diese Funktion wird auch vom Garbage-collector aufgerufen, da sie den „Java-Destruktor“ darstellt.

Fehlerbehandlung:

- Sollte eine Exception aus einer der beiden aufgerufenen Funktionen geworfen werden, wird ein Stacktrace erzeugt, und die Exception discarded, da Funktion auch vom Garbage-Collector aufgerufen wird.

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.19 **JNINetwork::addListener**

Prototyp:

- **public synchronized void** addListener(ChangeListener aListener)

Parameter:

- ChangeListener aListener: Ein javax.swing.event.ChangeListener, dessen StateChanged-Methode aufgerufen wird, wenn sich der Zustand des Objektes verändert.

Rückgabewert:

- **void**

Bemerkungen:

- Dem Socket kann ein Changelistener hinzugefügt werden, der informiert wird, sobald der Status des aktuellen Objektes sich ändert.
- Das **synchronized** ist hier eventuell etwas overkill, da lediglich die EventListenerListEx geschützt werden muss, und bereits Geschützt ist.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3.2.2.20 **JNINetwork::triggerStateChange**

Prototyp:

- **private synchronized void** triggerStateChange(**boolean** state)

Parameter:

- **boolean** state: Der neue Zustand. Folgende Werte sind definiert:
 - **private static final boolean** DISCONNECTED = **false**
 - **private static final boolean** CONNECTED = **true**

Rückgabewert:

- **void**

Bemerkungen:

- Die Methode setzt den neuen Status und ruft für alle registrierten EventListener die StateChanged Methode auf, indem die Methode **EventListenerListEx::fireStateChanged()** aufgerufen wird.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.2.2.21 JNINetwork::isConnected

Prototyp:

- `public boolean isConnected()`

Parameter:

- `void`

Rückgabewert:

- `boolean`: Der aktuell mit `JNINetwork::triggerStateChange()` gesetzte Status.
Mögliche Werte sind:

- `private static final boolean DISCONNECTED = false`
- `private static final boolean CONNECTED = true`

Bemerkungen:

- Die Methode setzt den neuen Status und ruft für alle registrierten EventListener die `StateChanged` Methode auf, indem die Methode `EventListenerListEx::fireStateChanged()` aufgerufen wird.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines, da die Klasse abstrakte Basisklasse ist.

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.2.2.22 JNINetwork::Miscellaneous

Die folgenden Funktionen sind des weiteren definiert und stellen die Schnittstelle zur Native Implementierung (siehe: Java Seite - Funktionsbeschreibung) dar. Aus den Signaturen werden mit javah -jni die notwendigen Signaturen der C-Funktionen erstellt.

```
/** This function just describes the Signature of the Native Code
 */
private native int initServerSocket(String aDottedIPAddress, short
aPort) throws JNINetworkException;

/** This function just describes the Signature of the Native Code
 */
private native int awaitConnection(int aServerSocket) throws
JNINetworkException;

/** This function just describes the Signature of the Native Code
 */
private native int openConnection(String aDottedIPAddress, short
aPort) throws JNINetworkException;

/** This function just describes the Signature of the Native Code
 */
private native void closeSocket(int aBufferID) throws
JNINetworkException;

/** This function just describes the Signature of the Native Code
 */
private native String recieve(int aBufferID) throws
JNINetworkException;

/** This function just describes the Signature of the Native Code
 */
private native void send(String aMessage, int aBufferID) throws
JNINetworkException;
```

10.3.2.2.23 FDZClientSocket::FDZClientSocket extends JNINetwork

Prototyp:

- **public** FDZClientSocket (InetAddress aServerAdress, **short** aServerPort, ChangeListener aChangeListener)
- **public** FDZClientSocket (InetAddress aServerAdress, **short** aServerPort)

Parameter:

- InetAddress aServerAdress: Wird an JNINetwork - Konstruktoren weitergeleitet
- **short** aServerPort: Wird an JNINetwork - Konstruktoren weitergeleitet

Optional ChangeListener aChangeListener: Wird an JNINetwork::addListener() weitergeleitet

Rückgabewert:

- FDZClientSocket: Eine Instanz der Klasse

Bemerkungen:

- Diese beiden Konstruktoren leiten die Parameter an die Basisklasse JNINetwork weiter.
- Diese Klasse ist die Konkrete Implementierung eines ClientSockets, also eines Sockets, der sich nur zu einem ServerSocket verbinden kann. Siehe Das Kapitel client in der Kommunikations-Bibliothek.
- **ACHTUNG!!!** Der Konstruktor ohne optionalen Parameter versucht sofort eine Verbindung zu öffnen. Dies ist ein nicht behobenes Überbleibsel aus früheren Code-Versionen und wurde nicht mehr behoben, da die Zeit zum Testen dieser Änderung fehlt.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.3.2.2.24 FDZClientSocket::recieve

Prototyp:

- `public String recieve() throws FDZNnetworkException`

Parameter:

- `void`

Rückgabewert:

- `String` Die empfangene Nachricht.

Bemerkungen:

- Diese Methode ruft lediglich `JNINetwork::recieve()` der Basisklasse auf.

Fehlerbehandlung:

- Wenn zunächst wird mit `JNINetwork::isConnected()` getestet, ob eine Verbindung besteht. Wenn keine Verbindung besteht, wird `JNINetwork::openConnection()` aufgerufen.
- Sollte `JNINetwork::recieve()` eine Exception werfen, so wird diese unverändert weitergeworfen.
- Anmerkung: Diese Stelle in der Anwendung würde die Möglichkeit einer einheitlichen Fehlerbehandlung bereitstellen, vergleiche `FDZClientSocket::send()`

Beispiele:

- Keines

10.3.2.2.25 FDZClientSocket::send

Prototyp:

- `public void send(String aMessage) throws FDZNetworkException`

Parameter:

- `String aMessage` Die zu sendene Nachricht

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode ruft lediglich `JNINetwork::send()` der Basisklasse auf.

Fehlerbehandlung:

- Wenn zunächst wird mit `JNINetwork::isConnected()` getestet, ob eine Verbindung besteht. Wenn keine Verbindung besteht, wird `JNINetwork::openConnection()` aufgerufen.
- Sollte `JNINetwork::recieve()` beim ersten Sendeversuch eine Exception werfen, so wird die Verbindung mit `JNINetwork::closeSocket()` abgebaut und sofort neu mit `FDZClientSocket::openConnection()` aufgebaut. Sollte eine dieser beiden Methoden eine Exception werfen, wird diese weitergeworfen, da dann ein tieferes Problem vorliegt (z.B. Gegenstelle ist nicht mehr Aktiv).
Wenn beim Neuaufbau der Verbindung nichts schief geht, wird die Nachricht erneut gesendet.
- Sollte `JNINetwork::recieve()` beim zweiten Sendeversuch eine Exception werfen, so wird diese auch weitergeworfen, um einen Endlos-loop zu verhindern. Die Anwendung muss somit entscheiden, wie weiter verfahren wird.
- Anmerkung: Diese Stelle in der Anwendung wurde die Möglichkeit genutzt, eine einheitliche Fehlerbehandlung zu implementieren. Um dies zu verbessern, müsste man den Grund der Exception überprüfen und entsprechend handeln.

Beispiele:

- Keines

10.3.2.2.26 FDZClientSocket::openConnection

Prototyp:

- `public void openConnection() throws FDZNException`

Parameter:

- `void`

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode ruft lediglich `JNINetwork::openConnection()` der Basisklasse auf.
- Diese Methode ist nur implementiert, um eine Meldung auszugeben, wenn in `JNINetwork ::openConnection()` eine Exception aufgetreten ist.

Fehlerbehandlung:

- Wenn in `JNINetwork::openConnection()` eine Exception aufgetreten ist, wird eine Meldung ausgegeben und die Exception wird weitergeworfen.

Beispiele:

- Keines

10.3.2.2.27 FDZServerSocket::FDZServerSocket extends JNINetwork

Prototyp:

- **public** FDZServerSocket (**short** aServerPort, ChangeListener aChangeListener)
- **public** FDZServerSocket (**short** aServerPort)

Parameter:

- **short** aServerPort: Wird an JNINetwork - Konstruktoren weitergeleitet

Optional ChangeListener aChangeListener: Wird an JNINetwork::addListener() weitergeleitet

Rückgabewert:

- FDZServerSocket: Eine Instanz der Klasse

Bemerkungen:

- Diese beiden Konstruktoren leiten die Parameter an die Basisklasse JNINetwork weiter.
- Diese Klasse ist die Konkrete Implementierung eines ServerSockets, also eines Sockets, zu dem sich ein ClientSocket verbinden kann. Siehe Das Kapitel server in der Kommunikations-Bibliothek.
- Beide Konstruktoren rufen JNINetwork::initServerSocket() auf.

Fehlerbehandlung:

- Sollte JNINetwork::initServerSocket() eine Exception werfen, wird eine Meldung ausgegeben, ein Stacktrace erzeugt, und die Exception discarded. Wenn die Exception nicht discarded würde, müsste die Anwendung einen unschönen Try-Catch-Block um den **new FDZServerSocket**-Aufruf legen.
- **ACHTUNG!!!** Das Discarden der Exception führt dazu, beim nächsten Aufruf von FDZServerSocket::recieve() oder FDZServerSocket::send() eine FDZNetworkException geworfen wird, oder, dass aus JNINetwork::awaitConnection() eine RuntimeException geworfen wird, wenn der Socket nicht inzwischen manuell JNINetwork::initServerSocket() aufgerufen wurde.
- **ToDo: Dieses Verhalten sollte verbessert werden. Dazu muss insbesondere eine Methode in JNINetwork erstellt werden, die eine Abfrage erlaubt, ob der initialisiert ist.!**

Beispiele:

- Keines

10.3.2.2.28 FDZServerSocket::recieve

Prototyp:

- `public String recieve() throws FDZNetworkException`

Parameter:

- `void`

Rückgabewert:

- `String` Die empfangene Nachricht.

Bemerkungen:

- Diese Methode ruft lediglich `JNINetwork::recieve()` der Basisklasse auf.

Fehlerbehandlung:

- Zunächst wird mit `JNINetwork::isConnected()` getestet, ob eine Verbindung besteht. Wenn keine Verbindung besteht, wird `JNINetwork::awaitConnection()` aufgerufen. Eine eventuelle Exception aus `JNINetwork::awaitConnection()` wird weitergeworfen.
- Sollte `JNINetwork::recieve()` eine Exception werfen, so wird diese unverändert weitergeworfen.
- Anmerkung: Diese Stelle in der Anwendung würde die Möglichkeit einer einheitlichen Fehlerbehandlung bereitstellen, vergleiche `FDZClientSocket::send()`

Beispiele:

- Keines

10.3.2.2.29 FDZServerSocket::send

Prototyp:

- `public void send(String aMessage) throws FDZNetworkException`

Parameter:

- `String aMessage` Die zu sendene Nachricht

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode ruft lediglich `JNINetwork::send()` der Basisklasse auf.

Fehlerbehandlung:

- Wenn zunächst wird mit `JNINetwork::isConnected()` getestet, ob eine Verbindung besteht. Wenn keine Verbindung besteht, wird `JNINetwork::openConnection()` aufgerufen.
- Sollte `JNINetwork::recieve()` beim ersten Sendeversuch eine Exception werfen, so wird die Verbindung mit `JNINetwork::closeSocket()` abgebaut und sofort neu mit `FDZServerSocket::awaitConnection()` aufgebaut. Sollte eine dieser beiden Methoden eine Exception werfen, wird diese weitergeworfen, da dann ein tieferes Problem vorliegt (z.B. Gegenstelle ist nicht mehr Aktiv, oder das Problem im Konstruktor von `FDZServerSocket`).
Wenn beim Neuaufbau der Verbindung nichts schief geht, wird die Nachricht erneut gesendet.
- Sollte `JNINetwork::recieve()` beim zweiten Sendeversuch eine Exception werfen, so wird diese auch weitergeworfen, um einen Endlos-loop zu verhindern. Die Anwendung muss somit entscheiden, wie weiter verfahren wird.
- Anmerkung: Diese Stelle in der Anwendung wurde die Möglichkeit genutzt, eine einheitliche Fehlerbehandlung zu implementieren. Um dies zu verbessern, müsste man den Grund der Exception überprüfen und entsprechend handeln.

Beispiele:

- Keines

10.3.2.2.30 FDZServerSocket::closeServerSocket

Prototyp:

- `public void closeServerSocket () throws FDZNetworkException`

Parameter:

- `void`

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode ruft `JNINetwork::closeServerSocket()` der Basisklasse auf. Durch wird lediglich die in der Basisklasse protected-te Funktion public.

Fehlerbehandlung:

- Keins

Beispiele:

- Keines

10.3 Java-Schnittstelle der Kommunikations-Bibliothek - Funktionsbeschreibung

10.3.2.2.31 FDZServerSocket::awaitConnection

Prototyp:

- **public void** awaitConnection() **throws** FDZNetworkException

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Diese Methode ruft JNINetwork::awaitConnection() der Basisklasse auf. Dadurch wird lediglich die in der Basisklasse protected-te Funktion public.

Fehlerbehandlung:

- Keins

Beispiele:

- Keines

10.3.2.2.32 FDZServerSocket::closeSocket

Prototyp:

- `public void closeSocket () throws FDZNetworkException`

Parameter:

- `void`

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode ruft `JNINetwork::closeSocket()` der Basisklasse auf. Dadurch wird lediglich die in der Basisklasse protected-te Funktion public.

Fehlerbehandlung:

- Keins

Beispiele:

- Keines

10.3.2.2.33 **JNINetworkException**::*JNINetworkException extends IOException*

Prototyp:

- **public** JNINetworkException (String aReason)
- **public** JNINetworkException (**int** aBufferID, **int** aNetworkErrorCode,
String aMessage, String aErrorCodeDescription)

Parameter:

- String aReason: Ein Fehlertext. Dieser Konstruktor ist nur eine Fall-Backlösung, falls die eingentlichen Daten der Exception nicht vorhanden sind.
- **int** aBufferID: SocketDescriptor des Sockets, auf dem der Fehler aufgetreten ist. Siehe **int** aConnID in HandlerType
- **int** aNetworkErrorCode: SocketDescriptor des Sockets, auf dem der Fehler aufgetreten ist. Siehe **int** aErr_no in HandlerType
- String aMessage: Eine von der Kommunikations-Bibliothek erzeugt Beschreibung des Fehlers. Siehe **const char*** aMsg in HandlerType
- String aErrorCodeDescription: System-Fehlernachricht, die zum angegebenen Fehlercode err_no gehört. Siehe **const char*** aErr_no_msg in HandlerType

Rückgabewert:

- JNINetworkException: Eine Instanz der Klasse

Bemerkungen:

- Diese Klasse wird verwendet, um die Informationen einer Fehlersituation in einer der Funktionen des Native Teils der jniBridge als Exception ins Java zu werfen. Siehe Funktionen in Native Seite - Funktionsbeschreibung.
- Die Klasse enthält noch accessor-Methoden für die Member der Klasse; diese sind so trivial, dass sie hier nicht weiter beschrieben werden.

Fehlerbehandlung:

- Keines, dies ist eine Exception

Beispiele:

- Keines

10.3.2.2.34 FDZNetworkException::FDZNetworkException *extends
JNINetworkException*

Prototyp:

- **public** FDZNetworkException(JNINetworkException aException)

Parameter:

- String aException: Eine JNINetworkException, um die diese Exception herumgelegt wird.

Rückgabewert:

- FDZNetworkException: Eine Instanz der Klasse

Bemerkungen:

- Diese Klasse wird verwendet, um JNINetworkException's im Java weiterzuverwenden. Innerhalb der Java-Seite werden nur FDZNetworkException's erzeugt.

Fehlerbehandlung:

- Keines, dies ist eine Exception

Beispiele:

- Keines

10.3.2.2.35 EventListenerListEx::EventListenerListEx
EventListenerList

extends

Prototyp:

- **public** EventListenerListEx(**void**)

Parameter:

- **void**

Rückgabewert:

- EventListenerListEx: Eine Instanz der Klasse

Bemerkungen:

- Diese Klasse erweitert javax.swing.event.EventListenerList um die Möglichkeit, für alle registrierten ChangeListener die Methode stateChanged aufzurufen.

Fehlerbehandlung:

- Keines, dies ist eine Exception

Beispiele:

- Keines

10.3.2.2.36 EventListenerListEx::addListener

Prototyp:

- **public synchronized void** addListener (ChangeListener aListener)

Parameter:

- ChangeListener aListener: Ein ChangeListener, der der Klasse hinzugefügt werden soll.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Methode ruft lediglich die `add`-Methode der Basisklasse mit den notwendigen Parametern auf.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.3.2.2.37 EventListenerListEx::fireStateChanged

Prototyp:

- `public synchronized void fireStateChanged(Object aSource)`

Parameter:

- `Object aSource`: Mit diesem Objekt wird ein ChangeEvent erzeugt. An die `stateChanged`-Methode des in `EventListenerListEx::addListener()` gesetzten ChangeListener wird dieses ChangeEvent übergeben. Mit `ChangeEvent.getSource()` kann `aSource` wieder extrahiert werden, um festzustellen, auf welchem Objekt sich der State geändert hat.

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode ruft die `stateChanged`-Methode eines jeden mit `EventListenerListEx::addListener()` hinzugefügten ChangeListener auf.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1 MessageHandler - Funktionsbeschreibung

10.4.1.1 Java-Seite

10.4.1.1.1 *JNIMessageCallback::JNIMessageCallback*

Prototyp:

- **public interface** JNIMessageCallback

Parameter:

- keiner Dies ist ein Interface

Rückgabewert:

- keiner Dies ist ein Interface

Bemerkungen:

- Dieses Interface definiert eine Object, das als Callback registriert werden kann.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4.1.1.2 `JNIMessageCallback::getRegisteredMessage`

Prototyp:

- `public String getRegisteredMessage()`

Parameter:

- `void`

Rückgabewert:

- `String` Die Nachricht, für die der Callback registriert werden soll

Bemerkungen:

- Diese Methode wird von `Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerJNIMessageCallback()` verwendet: Der String, der zurückgegeben wird, wird in `JNIMessageHandler::registerCallback()` verwendet, um den Callback zu registrieren.
- **ACHTUNG!!!** Der String darf sich während der Lebzeit des Objektes, das `interface JNIMessageCallback` implementiert, nicht ändern!!! Im Interface darf die Methode nur leider nicht so deklariert werden, dass sie ein `final` Member zurückgibt...

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.3 *JNIMessageCallback::messageRecieved*

Prototyp:

- **public void** messageRecieved(String message) ;

Parameter:

- String message: Der String, aufgrund dessen der Callback ausgelöst wurde.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Methode wird von callbackWrapper() aufgerufen, wenn der von JNIMessageCallback::getRegisteredMessage() zurückgegeben wurde einen Callback ausgelöst hat.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.4 **JNIMessageHandler::JNIMessageHandler extends JNIBridge**

Prototyp:

- **public abstract class JNIMessageHandler extends JNIBridge**

Parameter:

- **void**

Rückgabewert:

- **JNIMessageHandler** Eine Instanz der Klasse

Bemerkungen:

- Diese Klasse bildet die Grundlage der Java-seitigen Implementierung des MessageHandlers.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.5 JNIMessageHandler::registerCallback

Prototyp:

- **protected static void** registerCallback (JNIMessageCallback aDelegate)

Parameter:

- JNIMessageCallback aDelegate ein JNIMessageCallback

Rückgabewert:

- **void**

Bemerkungen:

- Mit dieser Methode wird ein Callback auf eine Java-Methode registriert. Diese Funktion ruft lediglich (wenn auch über eine dazwischen liegende Funktion Java_jniBridge_fdzMessageHandler_JNIMessageHandler_registerJNICallback ()) auf. Alles weitere siehe dort.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.6 *JNIMessageHandler::unregisterCallback*

Prototyp:

- **protected static void** unregisterCallback (JNIMessageCallback aDelegate)

Parameter:

- JNIMessageCallback aDelegate ein JNIMessageCallback

Rückgabewert:

- **void**

Bemerkungen:

- Mit dieser Methode wird ein Callback de-registriert. Vergleiche Java_jniBridge_fdzMessageHandler_JNIMessageHandler_unregisterJNICallback ()

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.7 JNIMessageHandler::dispatchMessage

Prototyp:

- `protected static int dispatchMessage(String aMessage)`

Parameter:

- `String aMessage` ein Nachricht, die geparsed werden soll

Rückgabewert:

- `int` die Anzahl gefundener und ausgeführter Callbacks

Bemerkungen:

- Mit dieser Methode wird Nachricht geparsed. Vergleiche `Java_jniBridge_fdzMessageHandler_JNIMessageHandler_dispatchJNIMessage()`

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.8 FDZMessageHandler::FDZMessageHandler extends JNIMessageHandler

Prototyp:

- `public abstract class JNIMessageHandler extends JNIBridge`

Parameter:

- `void`

Rückgabewert:

- FDZMessageHandler Eine Instanz der Klasse

Bemerkungen:

- Dies ist nur eine Abstraktionsschicht.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.9 FDZMessageHandler::registerCallback

Prototyp:

- **public static void** registerCallback (JNIMessageCallback aDelegate)

Parameter:

- JNIMessageCallback aDelegate **ein** JNIMessageCallback

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist nur eine Abstraktionsschicht. Ruft lediglich JNIMessageHandler::registerCallback () auf.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.10 FDZMessageHandler::unregisterCallback

Prototyp:

- **public static void** unregisterCallback (JNIMessageCallback aDelegate)

Parameter:

- JNIMessageCallback aDelegate ein JNIMessageCallback

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist nur eine Abstraktionsschicht. Ruft lediglich JNIMessageHandler::unregisterCallback() auf.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

10.4.1.1.11 FDZMessageHandler::dispatchMessage

Prototyp:

- `public static int dispatchMessage (String aMessage)`

Parameter:

- `String aMessage` ein Nachricht, die geparsed werden soll

Rückgabewert:

- `int` die Anzahl gefundener und ausgeführter Callbacks

Bemerkungen:

- Dies ist nur eine Abstraktionsschicht. Ruft lediglich `JNIMessageHandler::dispatchMessage ()` auf.

Fehlerbehandlung:

- Keines

Beispiele:

- Keines

10.4 MessageHandler

Im Folgenden wird die C++ - seitige Implementierung des MessageHandlers im Detail beschrieben.

10.4.1.2 C - Seite

10.4.1.2.1 messageHandler.c::SCallbackData

Prototyp:

```
• typedef struct _SCALLBACKDATA
{
    SMessage *message;
    //Custom Data that can be used by the caller of registerCallback.
    This pointer will be passed to the registered
    Callbackfunction. messageHandler is guaranteed no to use this
    data for its own purposes
    void *customData;
} SCallbackData;
```

Parameter:

- SMessage *message: Pointer auf die SMessage, die Registriert ist.
- **void** *customData: **void**-Pointer auf Daten, die an die Registrierte Callback-Funktion zurückgegeben wird. Dieser Pointer wird vom MessageHandler garantiert nicht verwendet oder verändert.

Rückgabewert:

- SCallbackData Kein eigentlicher Rückgabewert; erzeugt eine Struktur mit dem Typ.

Bemerkungen:

- Diese Struktur wird verwendet, um zum Aufruf eines Callbacks verwendeten Daten zu speichern. Vergleich `callbackWrapper()` und `parseMessage()`.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.4 MessageHandler

10.4.1.2.2 messageHandler.c::callback

Prototyp:

- `typedef void (*callback) (SCallbackData*)`

Parameter:

- `SCallbackData*`: Funktionsargument ist vom Typ Pointer-auf-`SCallbackData`.

Rückgabewert:

- `*callback` Kein eigentlicher Rückgabewert; erzeugt einen Function-Pointer mit dem Typ „Rückgabewert `void`, Übergabeparameter Pointer-auf-`SCallbackData`“.

Bemerkungen:

- Diese Struktur wird verwendet, um einen Pointer auf die Registrierte Callbackfunktion in `sMessageCallback` zu speichern. Alle Funktionen, die als Callbacks registriert werden sollen, müssen ebenfalls diese Signatur haben.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.4 MessageHandler

10.4.1.2.3 messageHandler.c::sMessageCallback

Prototyp:

```
• typedef struct _SMESSAGECALLBACK
{
    SCallbackData message;
    callback callbackFunction;
} sMessageCallback;
```

Parameter:

- SCallbackData message: Pointer auf die SCallbackData-Struktur, die Registriert ist.
- callback callbackFunction: callback (Also pointer auf Funktion), die aufgerufen werden soll.

Rückgabewert:

- sMessageCallback Kein eigentlicher Rückgabewert; erzeugt eine Struktur mit dem Typ.

Bemerkungen:

- Diese Struktur wird verwendet, um die registrierten Callbackfunktion zu speichern. Vergleiche registerCallback()

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.4 MessageHandler

10.4.1.2.4 messageHandler.c::testPrint

Prototyp:

- **void** testPrint (**void**)

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist lediglich ein Routine, die debugausgaben erzeugen kann.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.4 MessageHandler

10.4.1.2.5 messageHandler.c::registerCallback

Prototyp:

- **int** registerCallback(callback callbackFunction, SCallbackData * aMessage)

Parameter:

- **callback callbackFunction**: Die Funktion, die aufgerufen soll, wenn aMessage mit einer an `parseMessage()` übergebenen Nachricht übereinstimmt.
- **SCallbackData *aMessage** ein Pointer auf `SCallbackData`, das die Nachricht enthält, für die die `callbackFunction` registriert werden soll.

Rückgabewert:

- **int** Immer 0.

Bemerkungen:

- In dieser Funktion wird das übergebene `SCallbackData` in eine globales Array kopiert. Siehe `parseMessage()`.

Fehlerbehandlung:

- Keins

Beispiele:

- Demo des MessageHandler

10.4 MessageHandler

10.4.1.2.6 messageHandler.c::unregisterCallback

Prototyp:

- **int** unregisterCallback(callback callbackFunction, SCallbackData * aMessage)

Parameter:

- **callback callbackFunction**: Die Funktion, die aufgerufen soll, wenn aMessage mit einer an `parseMessage()` übergebenen Nachricht übereinstimmt.
- **SCallbackData *aMessage** ein Pointer auf SCallbackData, das die Nachricht enthält, für die die callbackFunction deregistriert werden soll.

Rückgabewert:

- **int** Immer 0.

Bemerkungen:

- Wenn beide Parameter dieser Funktion mit einem Eintrag in dem globalen Array übereinstimmen, wird die in `unregisterCallback()` erstellte Kopie des SCallbackData aus dem globalen Array entfernt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

10.4.1.2.7 messageHandler.c::parseMessage

Prototyp:

- `int parseMessage (SMessage *aMessage)`

Parameter:

- `SMessage *aMessage`: Die Nachricht als `SMessage`, die mit allen registrierten Callbacks verglichen werden soll.

Rückgabewert:

- `int` Die Anzahl der Callbacks, die mit `SMessage` registriert und somit ausgeführt worden sind.

Bemerkungen:

- Diese Funktion vergleicht vermittels der Funktion `compareMessage()` alle mit der im zweiten Parameter von `registerCallback()` registrierten Nachrichten mit der `SMessage aMassage`. Wenn die beiden übereinstimmen, wird der im ersten Parameter von `registerCallback()` hinterlegte `callback` ausgeführt und der interne Zähler um eins erhöht.

Fehlerbehandlung:

- Keins

Beispiele:

- Demo des MessageHandler

10.4.1.2.8 messageHandler.c::compareMessage

Prototyp:

- `int compareMessage (char *message1, char *message2)`

Parameter:

- `char *message1` Der eine Null-terminierte String, der verglichen werden soll
- `char *message2` Der andere Null-terminierte String, der verglichen werden soll

Rückgabewert:

- `int` 0, wenn die beiden Nachrichten „gleich“ (siehe Anmerkung) sind. 1 sonst.

Bemerkungen:

- Diese Funktion vergleicht zwei Nachrichten in ihrer Nullterminierten `char []`-Repräsentation.

Dabei werden folgende Wildcards unterstützt:

- Wenn beide Strings an jeder Stelle gleich sind, gelten die Strings als gleich.
- `*`: Ab der ersten Position, an der in einer der beiden Eingangsnachricht einer Stern auftaucht, gelten die beiden Strings als gleich.
- `?`: An genau der Position, an der in einer der beiden Eingangsnachricht ein Fragezeichen auftaucht, gelten die beiden Strings als gleich.

Ausserdem gilt: Wenn ein String länger als der andere ist, gelten die Strings als ungleich. Diese Logik wurde implementiert, um einheitliches Behandeln von Nachrichten zu ermöglichen.

Das Fragezeichen wird dabei typischerweise eingesetzt um das Timestamp-Feld zu maskieren.

Der Stern wird dabei typischerweise eingesetzt um Nachrichten mit vorher nicht bekannten Parametern zu maskieren (Palete).

Fehlerbehandlung:

- Keins

Beispiele:

- Demo des MessageHandler

10.4 MessageHandler

Listing 10.5: Demo des MessageHandler

```
// messageHandlerDevelop.c : Definiert den Einsprungpunkt für die
// Konsolenanwendung.
//

#include "stdafx.h"
#include "messageHandler.h"

//Diese Funktion wird immer als Callback-Funktion verwendet und dient
//nur der Ausgabe
//Außerdem wird eine mögliche Verwendung des customData-Pointers
//gezeigt
void testfunc(SCallbackData* aCallBackData)
{
    printf("Callback invoked by \"%s\" because it matched registered
           Message \"%s\"\n",
           aCallBackData->message->message,
           aCallBackData->customData);
}

int main(int argc, char* argv[])
{
    //Messages für Callbackregistrierung anlegen
    SMessage refmessage;
    SMessage refmessage1;

    SMessage testmessage1;
    SMessage testmessage2;
    SMessage testmessage3;

    //Ganz andere Message anlegen
    SMessage newMessage;

    SCcallbackData scd;

    //Vorbelegen der SMessages mit Strings
    initMessage(refmessage, "abcdefg");
    initMessage(testmessage1, "ab?defg");
    initMessage(testmessage2, "abc*");
    initMessage(testmessage3, "abcdefg");
    initMessage(newMessage, "Neue Nachricht");

    //SCallbackData - Struktur mit Message initialisieren
    scd.message = &testmessage1;
    //customData - Member auf registrierte Message erstellen, um in
    //testfunc() zu verwenden
    scd.customData = &testmessage1;
    //Callback mit der function void testfunc(SCallbackData*
    //aCallBackData) registrieren
    registerCallback(testfunc, &scd);

    //Wiederholen des gerade gesagten
```

10.4 MessageHandler

```
scd.message = &testmessage2;
scd.customData= &testmessage2;
registerCallback(testfunc, &scd);

scd.message = &testmessage3;
scd.customData = &testmessage3;
registerCallback(testfunc, &scd);

//Debug-Ausgabe
testPrint();

//Unterschiedliche Nachrichten parsen und Function aufzeigen.
parseMessage(&testmessage1);
parseMessage(&testmessage2);
parseMessage(&testmessage3);

//unregister
scd.message = &testmessage2;
unregisterCallback(testfunc, &scd);

//neues register
scd.message = &newMessage;
registerCallback(testfunc, &scd);

//Debug-Ausgabe
testPrint();

scd.message = &newMessage;
registerCallback(testfunc, &scd);

testPrint();

getc(stdin);
return 0;
}
```

10.5 JarExtractor

JarExtractor ist ein zusätzliches Modul, mit dem Image-Daten, die in einer jar-Datei gepackt wurden, extrahiert werden können. Weiterhin ist es möglich, gepackte Dateien zu extrahieren und neben der jar-Datei abzulegen. Dabei werden auch die Ordnerstrukturen in der jar-Datei berücksichtigt.

10.5.1 Konstruktor

- `public JarExtractor(String jarFileName)`
- Eine neue Instanz von JarExtractor wird erstellt. Bei `jarFileName` muß die Dateiendung mit angegeben werden.

10.5.2 Funktionen

10.5.2.1 getImage

Ein in der jar-Datei gepacktes Bild wird als Image zurückgegeben.

Prototyp:

- public Image getImage(String name);

Parameter:

- name : Der Name des Images, das verwendet werden soll.

Rückgabewert:

- Das mit name benannte Image, falls es in der jar-Datei vorhanden ist. Ansonsten null.

Bemerkungen:

- Keine

Fehlerbehandlung:

- Falls das geforderte Image nicht vorhanden ist, wird null zurückgegeben.

10.5 JarExtractor

10.5.2.2 getImageIcon

Ein in der jar-Datei gepacktes Bild wird als ImageIcon zurückgegeben.

Prototyp:

- public Image getImageIcon(String name);

Parameter:

- name : Der Name des Bildes, das verwendet werden soll.

Rückgabewert:

- Das mit name benannte Bild, falls es in der jar-Datei vorhanden ist. Ansonsten null.

Bemerkungen:

- Keine

Fehlerbehandlung:

- Falls das geforderte ImageIcon nicht vorhanden ist, wird null zurückgegeben.

10.5 JarExtractor

10.5.2.3 extractResource

Eine in der jar-Datei gepackte Datei wird neben die jar-Datei kopiert.

Prototyp:

- public int extractResource(String name);

Parameter:

- name : Der Name der Datei, die kopiert werden soll.

Rückgabewert:

- true, falls die mit name benannte Datei in der jar-Datei vorhanden ist und erfolgreich kopiert wurde. Ansonsten null.

Bemerkungen:

- Für Image-Daten nicht zu empfehlen (siehe Funktion getImage(String name)). Falls in der jar-Datei eine Dynamische Bibliothek oder ein shared object liegt, das per Java Native Interface verwendet werden soll, dann muß diese dll oder das so mit extractResource(String name) neben die jar-Datei kopiert werden.

Fehlerbehandlung:

- Falls die geforderte Datei nicht vorhanden ist oder beim Kopiervorgang ein Fehler passierte, wird null zurückgegeben. Bei einem erfolgreichen Kopiervorgang ist der Rückgabewert false;

10.5 JarExtractor

10.5.2.4 extractResource

Eine in der jar-Datei gepackte Datei wird neben die jar-Datei kopiert.

Prototyp:

- public int extractResource(String name, boolean directory);

Parameter:

- **name**: Der Name der Datei, die kopiert werden soll.
- **directory**: gibt an, ob die zu extrahierende Ressource ein Directory ist oder nicht

Rückgabewert:

- true, falls die mit name benannte Datei in der jar-Datei vorhanden ist und erfolgreich kopiert wurde. Ansonsten null.

Bemerkungen:

- Für Image-Daten nicht zu empfehlen (siehe Funktion getImage(String name)). Falls in der jar-Datei eine Dynamische Bibliothek oder ein shared object liegt, das per Java Native Interface verwendet werden soll, dann muß diese dll oder das so mit extractResource(String name) neben die jar-Datei kopiert werden.

Fehlerbehandlung:

- Falls die geforderte Datei nicht vorhanden ist oder beim Kopiervorgang ein Fehler passierte, wird null zurückgegeben. Bei einem erfolgreichen Kopiervorgang ist der Rückgabewert false;

10.5 JarExtractor

10.5.2.5 listEntries

Listet alle in der jar-Datei gepackten Datei auf dem Standard-Ausgabe-Stream auf.

Prototyp:

- public void listEntries();

Parameter:

- Keine

Rückgabewert:

- Keiner

Bemerkungen:

- Keine

Fehlerbehandlung:

- Keine.

10.5 JarExtractor

10.5.2.6 initialize

Initialisiert das JarExtractor-Objekt und wird lediglich im Konstruktor aufgerufen.

Prototyp:

- `private void initialize();`

Parameter:

- Keine.

Rückgabewert:

- Keiner

Bemerkungen:

- Alle in der jar-Datei gepackten Dateien werden in einer Hashtable gespeichert.

Fehlerbehandlung:

- Es wird eine `java.io.IOException` geworfen, falls die jar-Datei nicht vorhanden ist.

11 Lagersystem

11.1 Lager

11.1.1 Umsetzung

An dieser Stelle werden die Abläufe und Maßnahmen skizziert, die zur Lösung der beschriebenen Aufgaben umgesetzt werden. Es wird hier keine genaue Funktions- oder Ablaufbeschreibung des Codes geben, sondern es soll ein Verständnis für die Änderungen (was, welche Datei(en)/Funktion(en)) am Code geschaffen werden.

11.1.1.1 Entfernung des früheren GUI-Codes

Alle Funktionen zur Bearbeitung der Kommunikation wurde in der Datei `new_etherne*t` gekapselt. Die Funktionen und entsprechende Funktionsaufrufe in anderen Dateien zur Bearbeitung der GUI wurden entfernt.

11.1.1.2 Austausch TCP/IP Kommunikationsschicht

Im Rahmen der Integration des `fdzNetwork`-Moduls wurde die `new_etherne*t` funktionell weitestgehend entkernt, d.h. alle Zugriffe auf die alte API sowie `Talk_tcp.*` wurden entfernt. Die neue API des Kommunikationsmoduls wurde anschließend in die übrigen Funktionsrümpfe innerhalb der `new_etherne*t` zur Prüfung der Netzwerkkommunikation und zum Senden und Empfangen von Nachrichten gekapselt. Die Sende- und Empfangspuffer wurden global in der Hauptdatei `storage.c` angelegt. Die Netzwerkfehlerbehandlung wird in einem eigenen Punkt beschrieben. Die genaue Funktionsweise des allgemeinen Netzwerkmoduls ist der entsprechenden Dokumentation zu entnehmen. Eine genaue Beschreibung der neuen und angepassten Funktionen ist im späteren Verlauf dieses Dokuments enthalten.

11.1.1.3 Einsatz Kommandoparser

Alle Aufrufe der alten API zum Kommando generieren und parsen in `cmd_pars.*`, sowie die Dateien selbst wurden aus dem Projekt entfernt. Für den Einsatz des `fdzMessageHandler` wurden zwei Callbackfunktionen in der `lager.c` implementiert. Die Registrierung dieser Callbackfunktionen mit der entsprechenden Nachrichtensignatur findet im `main` der `storage.c` noch vor der Erstellung der beiden Tasks statt. Sollte diese Registrierung im Programmablauf fehlschlagen, wird die Software mit einer Fehlermeldung beendet. Das Auslösen eines Callbacks,

11.1 Lager

d.h. das Prüfen einer empfangenen Nachricht auf Korrektheit und Aufruf eines Callbacks bei zu treffender Nachrichtensignatur, findet nach Empfang einer Nachricht innerhalb der Datei `p_comm` statt. Weiterhin musste in der Datei `new_etherneet.c` eine Funktion zum globalen Zwischen-speichern des Timestamps der letzten empfangenen Nachricht implementiert werden, um protokollgerechte Quitierungen (mit Timestamp des Auftrags) erstellen zu können. Sollte beim Parsen der Nachricht im Modul `fdzMessageHandler` ein Fehler auftreten, wird das Lager heruntergefahren, da ein stabiler Lagerbetrieb nicht mehr gewährleistet werden kann.

11.1.2 Implementierung Kommandogenerator

Die Erstellung von Quitierungen wurde in der Datei `new_etherneet.c` implementiert. Die Nachricht wird global in den Sendepuffer eingetragen. Sie besteht aus einer Signatur, einem übergebenen Quittungsbuchstaben und Nummer und dem global gespeicherten Timestamp.

11.1.2.1 Fehlerbearbeitung und Recovery

- Fehlerbearbeitung

Die Fehlerbearbeitung seitens Hardware wurde mit einer Ausnahme komplett aus dem alten Stand übernommen. Fehler in der Lagerhardware werden nach wie vor in einer Fehlerqueue gespeichert und das Lager in einen Fehlerzustand versetzt, sollte der jeweilig auftretende Fehler nicht im Rahmen der Bearbeitung des aktuellen Schritts behoben werden können. Alle Hardwarefehler müssen über die Handsteuerung quittiert werden. Nach Quittierung eines Fehlers wird versucht das Lager in Grundstellung (Fach 1) zu fahren. Sollte beim Anfahren der Grundstellung erneut ein Fehler auftreten, wiederholt sich dieser Ablauf ab Abfrage der Handsteuerung. Sollte ein Fehler nicht behebbar sein, wird das Lager nach negativer Quittierung, ohne Fehlermeldung an die Lagersteuerung, sofort heruntergefahren. Ein in der ursprünglichen Version zusätzlich eingebauter Wartezustand im Falle einer negativen Quittierung wurde entfernt, da das System bei Tests dieses Zustands in einen Deadlock lief.

- Anpassung des Recovery an das neue Protokoll

Die Funktionalität des Recovery, d.h. speichern einer Nachricht nach positiver Quittierung (A001) bis zur positiven oder negativen Bearbeitungsquittierung (A002, F001, F002), ist in der Datei `recovery.c` bereits implementiert. Eine Nachricht wird nach wie vor in der Datei `RECOV.DAT` auf dem Lagerchip als String gespeichert. Beim Hochfahren und nach einer Grundstellungsfahrt infolge eines Hardwarefehlers wird die Datei ausgelesen und dem Operator zur Wahl gestellt, ob er den letzten Auftrag nochmals an die Hardware übergeben will. Bei erfolgreichem Abschluss des Auftrags wird der Inhalt der Datei gelöscht. Die Datei selbst bleibt erhalten. Die zu speichernde Länge der Nachricht musste entsprechend dem neuen Protokoll angepasst werden.

11.1.2.2 Beibehaltung Bearbeitungslogik durch zwei Tasks

Die Funktionen zur jeweiligen Taskbearbeitung wurden vollständig überarbeitet. Die Aufgaben der Tasks sind aber geblieben. Task 1 bearbeitet und überwacht auf Basis der übernommenen

11.1 Lager

API die Steuerung der Hardware, Task 2 überwacht den Nachrichtenempfang und steuert den Auftragsfortschritt bezüglich der Kommunikation mit der Lagersteuerung. Die beiden Tasks arbeiten aber nur bedingt parallel. Während der Bearbeitung des Auftrags durch die Lagerhardware in Task 1 prüft Task 2 nur den Zustand der Kommunikation zur Lagersteuerung und versucht bei Verbindungsabbruch diese wiederherzustellen. Die Taskkoordination, d.h. wann welcher Task aktiv arbeiten darf, wird über globale Flags gesteuert und in Kapitel ?? erläutert.

11.1.2.3 Integration der Lichtschanke zur Lagerfachprüfung

Der deplazierte und unvollständige Code zur Abfrage der Lichtschanke wurde zunächst entfernt und später wieder in die Schrittkettenbearbeitung des Ventils integriert.

11.1.2.4 Überarbeitung des Codes

Große Blöcke auskommentierten Codes sowie ungenutzte Funktionen und Variablen, wurden entfernt. Weiterhin wurden bei einigen ursprünglichen Funktionen Kommentare hinzugefügt, so weit der Sinn dieser Funktionen ersichtlich war.

11.1.3 Flagbeschreibung für Tasksteuerung und -Interaktion

Der Zustand des Lagers wird durch verschiedene Flags abgebildet. Diese Flags beschreiben den aktuellen Zustand der Kommunikation, der Lagerhardware, des aktuellen Auftrags und den aktiven Task. Auf Basis dieser globalen Flags wurde die Ablauflogik innerhalb und zwischen den Tasks realisiert. Hier soll eine Übersicht über diese Steuerflags und ein Einstieg zu den im weiteren Verlauf beschriebenen Ablaufbeschreibungen gegeben werden.

11.1.3.1 lagerBereit

Gibt an, welcher Task aktiv arbeitet. Die Parallelisierung der Abläufe wird über dieses Flag weitestgehend eingeschränkt. Es wird immer zu Beginn der beiden Tasks geprüft, um festzustellen, ob der jeweilige Task arbeiten darf und als letztes geschrieben, um einen Taskwechsel herbeizuführen.

11.1.3.1.1 Wert: 0

Gibt an, dass Task 1, d.h. die Steuerung der Lagerhardware arbeitet. Solange dies der Fall ist pollt Task 2 ausschließlich den Zustand der Verbindung über das Flag `netzwerkAktiv`, und versucht bei einem Verbindungsabbruch die Verbindung erneut herzustellen. Es werden weder neue Aufträge entgegengenommen, noch der aktive Auftrag in irgendeiner Weise an die Lagersteuerung zurückgemeldet. Wenn der Task 1 die Bearbeitung des Auftrags bezüglich der Hardware-Steuerung abgeschlossen hat, wird dieses Flag auf 1 gesetzt.

11.1 Lager

11.1.3.1.2 Wert: 1

Gibt an, dass Task 2, d.h. die Auftrags- und Netzwerksteuerung aktiv ist. Wenn ein Auftrag zur Bearbeitung im System existiert, wird versucht eine Antwort zur erfolgreichen / nicht erfolgreichen (abhängig vom global gesetzten Sendepuffer) Auftragsbearbeitung an die Lagersteuerung zu schicken. Wenn kein Auftrag aktiv ist, wird das Netzwerk auf eingehende Nachrichten geprüft. Bei Empfang eines Auftrags wird dieses Flag im Zuge der Übergabe der Auftragsdaten an die Hardwaresteuerung auf 0 gesetzt.

11.1.3.2 iModus

Beschreibung:

Ist in `storage.c` definiert und legt den Modus der Hardwaresteuerung in Task 1 fest.

11.1.3.2.1 MODUS_ERROR

Wird innerhalb Task 1 gesetzt, wenn in der Lagerhardware ein Fehler aufgetreten ist. In diesem Modus findet die Fehlerbearbeitung über das LCD und die Handsteuerung statt. Die Lagerhardware ruht, bis alle Fehler behandelt und der Modus auf MODUS_AUTO gesetzt wurde.

11.1.3.2.2 MODUS_DEBUG

Kann beim Start der Lagersoftware durch die Übergabe als 2. Parameter gesetzt werden. In diesem Zustand können über die Konsole direkte Kommandos zum Ein- und Auslagern von Paletten eingegeben werden. Da dieser Modus ohne Änderungen übernommen wurde, wird dieser im weiteren Verlauf nicht weiter beschrieben.

11.1.3.2.3 MODUS_AUTO

Standardmodus nach dem Hochfahren des Systems. In diesem Modus findet die gesamte reguläre Auftragsbearbeitung innerhalb der Lagerhardware (Task 1) statt. Bei Auftreten eines Fehlers in der Lagerhardware wird in den Modus MODUS_ERROR umgeschaltet.

11.1.3.2.4 MODUS_SHUTDOWN

11.1.3.3 recoveryFull

Beschreibung:

Gibt an, ob die Recovery-Datei `RECOV.DAT` gefüllt ist.

11.1 Lager

11.1.3.3.1 Wert: 0

Recovery ist leer. Wird nach dem Empfang und dem erfolgreichen Parsen einer Nachricht im Zuge des Schreibens der Recovery-Datei auf 1 gesetzt.

11.1.3.3.2 Wert: 1

Recovery ist voll. Wird beim Hochfahren des Systems überprüft. Ist dieses Flag gesetzt wird der Auftrag aus der Datei ausgelesen und entsprechend der Quittierung des Operators nochmals gestartet oder abgebrochen. Nach Senden der Bearbeitungsantwort (positiv: A002 oder negativ: F001/F002) wird es 0 gesetzt.

11.1.3.4 netzwerkAktiv

Beschreibung:

Gibt den Verbindungsstatus zur Lagersteuerung wieder.

11.1.3.4.1 Wert: 0

Es besteht keine Verbindung. Es wird versucht eine Verbindung aufzubauen. Bei Erfolg wird das Flag auf 1 gesetzt.

11.1.3.4.2 Wert: 1

Die Kommunikation zur Lagersteuerung ist möglich. Sollte beim Empfangen oder Senden von Nachrichten ein Fehler auftreten wird dieses Flag auf 0 gesetzt.

11.1.3.5 auftragAktiv

Beschreibung:

Gibt an, ob gerade ein Auftrag bearbeitet wird.

11.1.3.5.1 Wert: 0

Kein Auftrag aktiv. Bei Empfang eines Auftrags wird es auf 1 gesetzt.

11.1.3.5.2 Wert: 1

Ein Auftrag wird gerade bearbeitet. Wird bei erfolgreichem oder fehlerhaftem Bearbeitungsschluss auf 0 gesetzt.

11.1 Lager

11.1.3.6 task1run, task2run

Bedeutung:

Gibt an ob die beiden Tasks arbeiten oder beendet werden sollen.

11.1.3.6.1 Wert: 0

Das Lager soll heruntergefahren werden. Die Taskbearbeitung stoppt und es wird der für die Taskbearbeitung bereitgestellte Speicher freigegeben und anschließend das Programm beendet.

11.1.3.6.2 Wert: 1

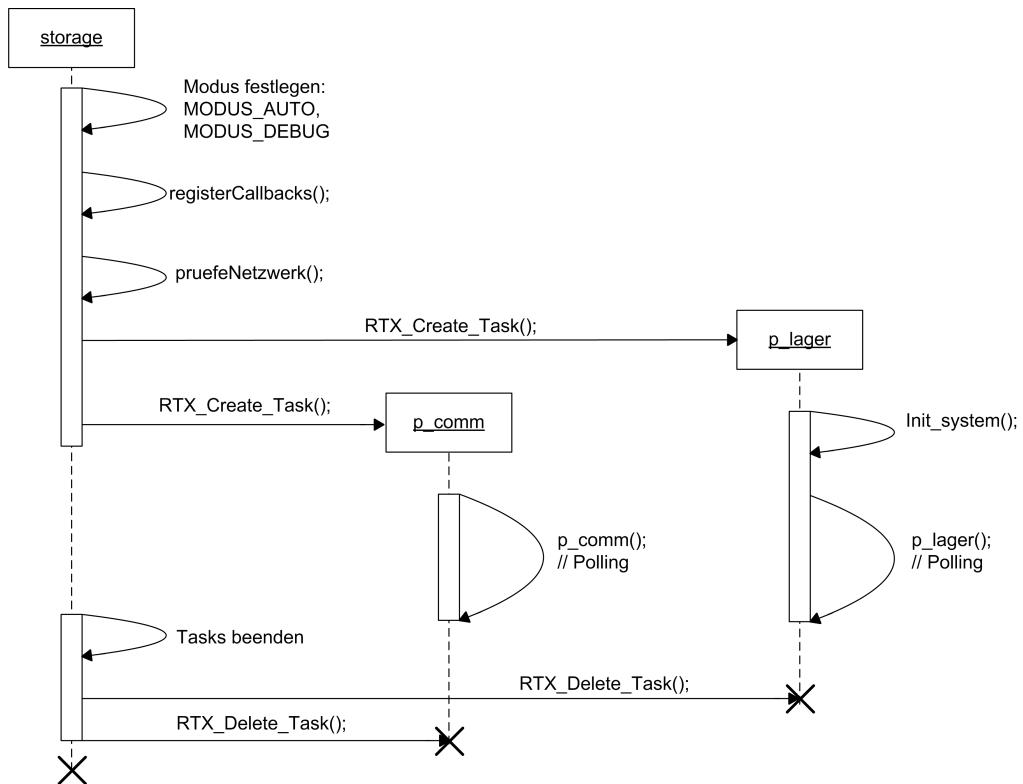
Lager soll arbeiten. Die Funktionen zum Bearbeiten der Tasks werden in Schleifen durchlaufen.

11.1.4 Ablaufbeschreibungen

An dieser Stelle werden mögliche Situationen während des Lagerbetriebs skizziert. Die Abläufe werden anhand der Koordination der Tasks untereinander und mit der Lagersteuerung erläutert, um die Verarbeitungslogik der Lagersoftware verständlicher zu machen. Die Funktion `p_lager()` bildet die Funktionalität zur Steuerung und Kontrolle der Hardware in Task 1 ab. Sie wird in einer Polling-Schleife immer wieder durchlaufen. Um festzustellen, ob die Hardware (re-)agieren muss wird innerhalb der Funktion zu Beginn das Flag `lagerBereit` auf 0 geprüft. Die Funktion `p_comm()` bildet die Funktionalität zur Kommunikation mit der Lagersteuerung in Task 2 ab. Sie wird in einer Polling-Schleife immer wieder durchlaufen. Um festzustellen, ob das eine Nachricht empfangen oder versendet werden muss wird innerhalb der Funktion zu Beginn das Flag `lagerBereit` auf 1 geprüft.

11.1 Lager

11.1.4.1 System hochfahren

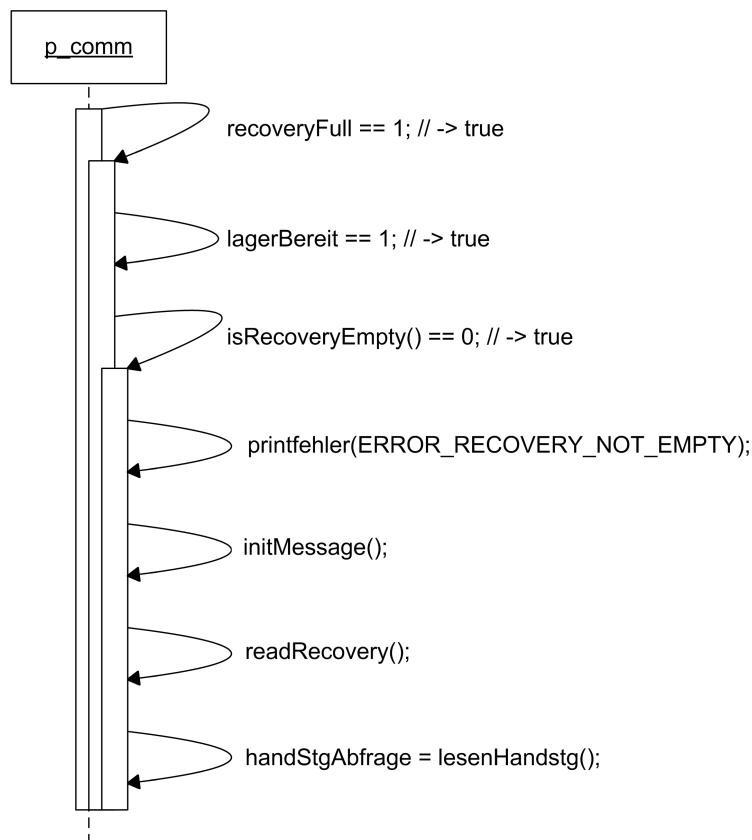


Beide Tasks werden über die main-Funktion in `storage.c` initialisiert und gestartet. Über die Kommandozeile kann das Lager optional im Debug-Modus gestartet werden, sonst wird es standardmäßig im Auto-Modus gestartet. Die folgenden Ablaufbeschreibungen beschränken sich ausschließlich auf die Modi `MODUS_AUTO` und `MODUS_ERROR`, da alle weiteren Modibearbeitungen unverändert übernommen wurden. Nach Start der Tasks, werden die Callbackfunktionen zur Ein- und Auslagerbearbeitung `cbfEinlagern()` und `cbfAuslagern()` für das Modul `fdzMessageParser` und die Fehlerfunktion `networkErrorHandler` des `fdzNetwork` Moduls über `registerCallbacks()` aus `new_ethernet.c` registriert. Weiterhin wird über `pruefeNetzwerk()` versucht eine Verbindung zur Lagersteuerung über die als Parameter 1 angegebene IP-Adresse und dem Port 30000 aufzubauen. Die Initialisierung des Lagers in `storage.c` endet mit der Erzeugung der Tasks 1 und 2. Ab diesem Zeitpunkt übernehmen diese die Steuerung des Lagers. In Task 1 wird vor dem Polling von `p_lager()` die Grundstellung des Lagers über `init_System()` initialisiert. Die Prüfung der Recovery-Datei erfolgt innerhalb von `p_comm()` ???. Danach kann das Lager sequentiell Aufträge entgegennehmen. Wenn im laufenden Betrieb ein nicht behebbarer Fehler in einem der beiden Tasks auftritt, wird das Lager nach Operatoranfrage heruntergefahren. Wann das Lager heruntergefahren werden soll, erkennt `storage.c` anhand der Flags `task1run` und `task2run`. Diese werden bei Betriebsabbruch in einem der beiden Tasks gesetzt. Eine genaue Fehlerbearbeitung über die Handsteuerung wird im weiteren Verlauf erläutert.

11.1 Lager

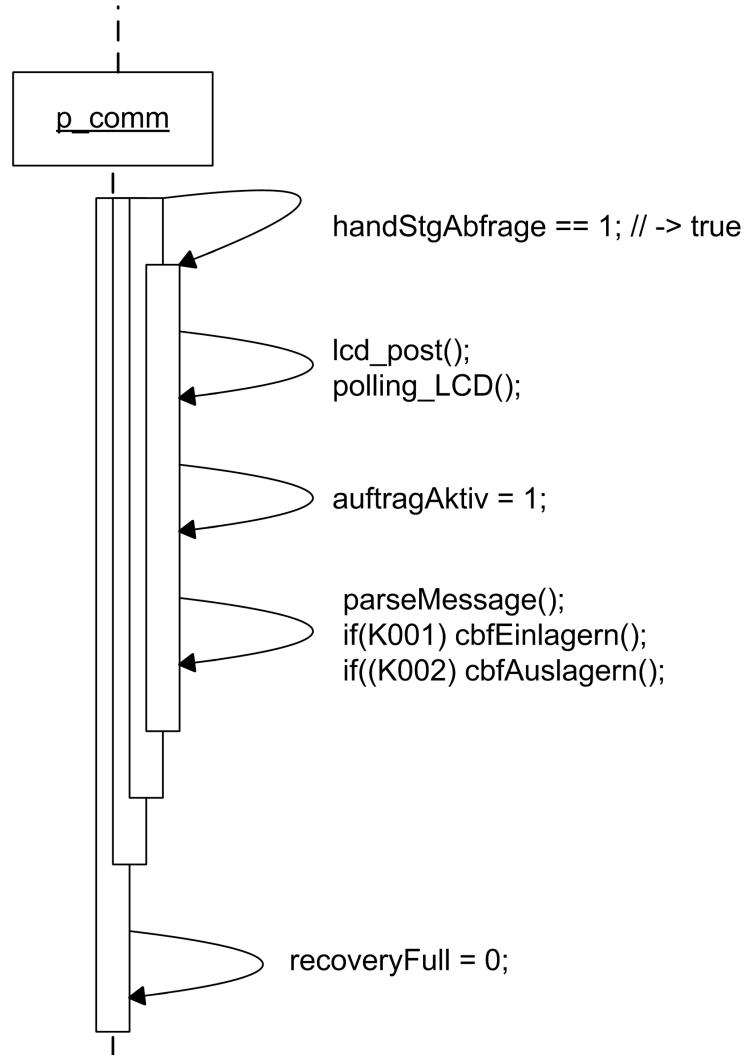
11.1.4.2 Recovery-Prüfung

11.1.4.2.1 Prüfung und Operatoranfrage



Zu Beginn des Task 1 wird das Flag `recoveryFull` geprüft. Wenn dieses gesetzt ist, wird weiterhin über das Flag `lagerBereit` geprüft, ob die Lagerhardware in der Lage ist einen Auftrag entgegenzunehmen. Ist dies gegeben, wird über `isRecoveryEmpty()` aus `recover.c` geprüft, ob die Datei Daten enthält. Wenn dem so ist, wird dies dem Operator über das LCD angezeigt. Weiterhin wird der globale Empfangspuffer initialisiert, der gespeicherte Auftrag aus der Datei RECOV.DAT ausgelesen und in den Puffer übertragen. Beim Auslesen der Datei wird der globale Antwortpuffer bereits mit dem zu Auftrag passenden Fehler beschrieben (F001 -> Fehler bei Einlagerung, F002 -> Fehler bei Auslagerung) vorbesetzt. Der Operator hat nun die Wahl den Auftrag zu wiederholen (ohne nochmalige Quittierung mit A001) oder den Auftrag abzubrechen.

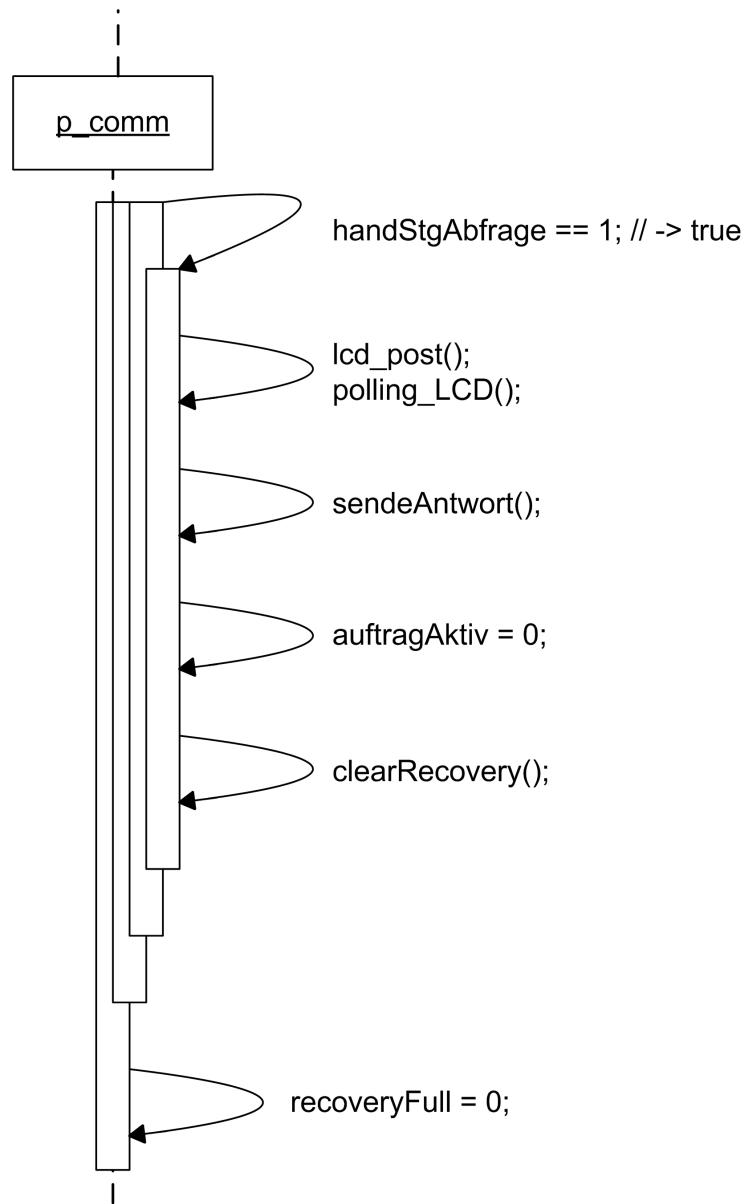
11.1.4.2.2 Auftragswiederholung



Bei Wiederholung wird über die Funktion `lcd_post()` der Ausgabetext des LCD auf IDLE gesetzt und über `polling_LCD()` ausgegeben. Beide Funktionen befinden sich in `error.c`. Anschließend wird das Flag `auftragAktiv` gesetzt, der globale Nachrichtenpuffer über die Funktion `parseMessage()` des Moduls `fdzMessageHandler` analysiert und die entsprechende Callbackfunktion aufgerufen. Durch die Analyse eines Einlagerauftrags (K001) wird die Funktion `cbfEinlagern()` aufgerufen, bei einem Auslagerauftrag (K002) wird die Funktion `cbfAuslagern()` aufgerufen. Beide Funktionen befinden sich in der Datei `lager.c`. Die weiteren Abläufe entsprechen einer Auftragsbearbeitung ohne Operatoreingriff nach `parseMessage()`.

11.1 Lager

11.1.4.2.3 Auftragsabbruch



Im Falle des Auftragsabbruchs wird über die Funktion `lcd_post()` der Ausgabetext des LCD auf IDLE gesetzt und über `polling_LCD()` ausgegeben. Beide Funktionen befinden sich in `error.c`. Anschließend wird über `sendeAntwort()` aus `new_etherenet.c` die global vorbelegte Fehlerquittung an die Lagersteuerung geschickt und der Inhalt der Recovery-Datei über `clearRecovery()` aus `recover.c` gelöscht. Abschließend wird das Flag `recoveryFull` zurückgesetzt. Das Lager ist danach in Bereitschaft, d.h. es kann Aufträge entgegennehmen. Die entsprechenden Abläufe werden im weiteren Verlauf beschrieben (siehe Kapitel ??)

11.1.4.3 Auftragsbearbeitung ohne Operatoreingriff

11.1 Lager

11.1.4.3.1 Auftragseingang



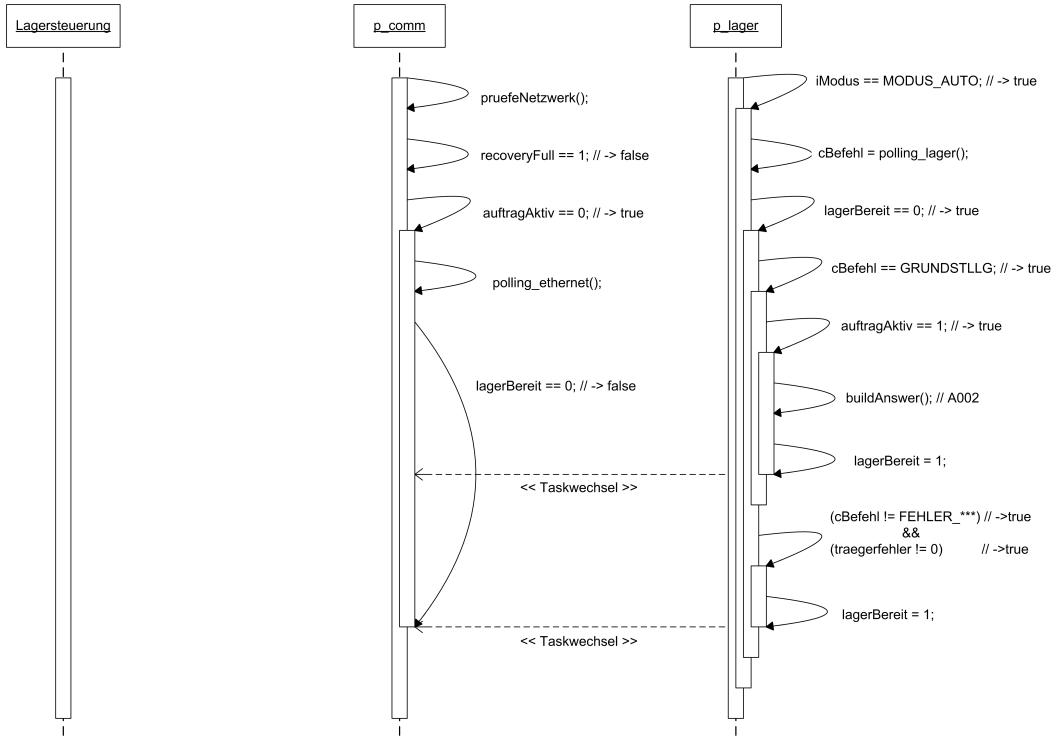
Ausgangspunkt einer Auftragsbearbeitung ohne Operatoreingriff aus dem Netzwerk ist ein fehlerfrei Hochgefahrenes System, so dass `lagerBereit` eins, `iModus` `MODUS_AUTO`, `auftragAktiv` null und `recoveryFull` null ist. Zu Beginn der Funktion `p_comm` wird der

11.1 Lager

Verbindungsstatus geprüft. Anschließend wird geprüft, ob das Flag zur Prüfung der Recovery-Datei gesetzt ist. Ist dem nicht der Fall wird weiterhin geprüft, ob ein Auftrag bereits aktiv ist. Ist dies nicht gegeben, wird das Netzwerk über `polling_etherneet()` auf eingehende Nachrichten geprüft. Wenn eine Nachricht eingeht, wird zunächst geprüft, ob `lagerBereit` nicht 0 ist, da sonst die Lagerhardware arbeiten würde. Wenn das Lager bereit ist, wird über `saveTimestamp()` der Zeitstempel der Nachricht global gesichert. Anschließend wird der globale Nachrichtenempfangspuffer über `parseMessage()` analysiert. Wenn ein Einlagerauftrag (K001) einging, wird `cbfEinlagern()` aufgerufen. Wenn ein Auslagerauftrag (K002) einging, wird `cbfAuslagern()` aufgerufen. Diese beiden Funktionen sind beinahe identisch. In ihnen wird zunächst das zu bearbeitende Fach mit `getFach()` extrahiert. Nur wenn `auftragAktiv` nicht gesetzt ist, d.h. kein Recoveryfall vorliegt und somit eine Quittung für den Auftragserhalt geschickt werden muss, wird der gesamte Nachrichtenstring über `saveRecovery()` in `RECOV.DAT` gesichert. Danach wird über `buildAnswer()` die Antwort `laSLA001` im globalen Sendepuffer erstellt und mit `sendeAntwort()` an die Lagersteuerung geschickt. Die weiteren Vorgänge gelten wieder unabhängig davon, ob ein Recovery-Fall vorliegt, oder nicht. Nun wird der globale Sendepuffer mit der dem Auftragstyp entsprechenden Fehlernachricht (F001 -> Einlagerung in `cbfEinlagern()`, F002 -> Auslagerung in `cbfAuslagern()`) vorbesetzt. Schließlich werden der Auftragstyp und das Fach über `lager()` aus `lager.c` an die Hardware übergeben und gestartet. Durch Setzen des Flags `auftragAktiv` wird für die weitere Bearbeitung festgelegt, dass kein Auftrag mehr empfangen werden kann, sondern nach dessen Bearbeitung eine Quittung geschickt werden muss. Abschließend wird durch Setzen des Flags `lagerBereit` auf null der Taskwechsel initiiert. Während der Aktivitäten des Tasks 1 in `p_comm()` wird im Task 2 in `p_lager()` der Modus geprüft. Im `MODUS_AUTO` wird dann über `polling_lager()` der physikalische Zustand des Lagers abgefragt. Sollten hier bereits Fehler auftreten, wird in `MODUS_ERROR` umgeschaltet (Beschreibung Kapitel ??). Weiterhin wird `lagerBereit` geprüft. Solange dieses Flag eins ist wiederholt der Task diesen Ablauf ohne weitere Aktionen. Nach Wechsel des Flags auf eins findet die Auftragsbearbeitung und -kontrolle durch die Lagerhardware statt (siehe nächster Punkt)

11.1 Lager

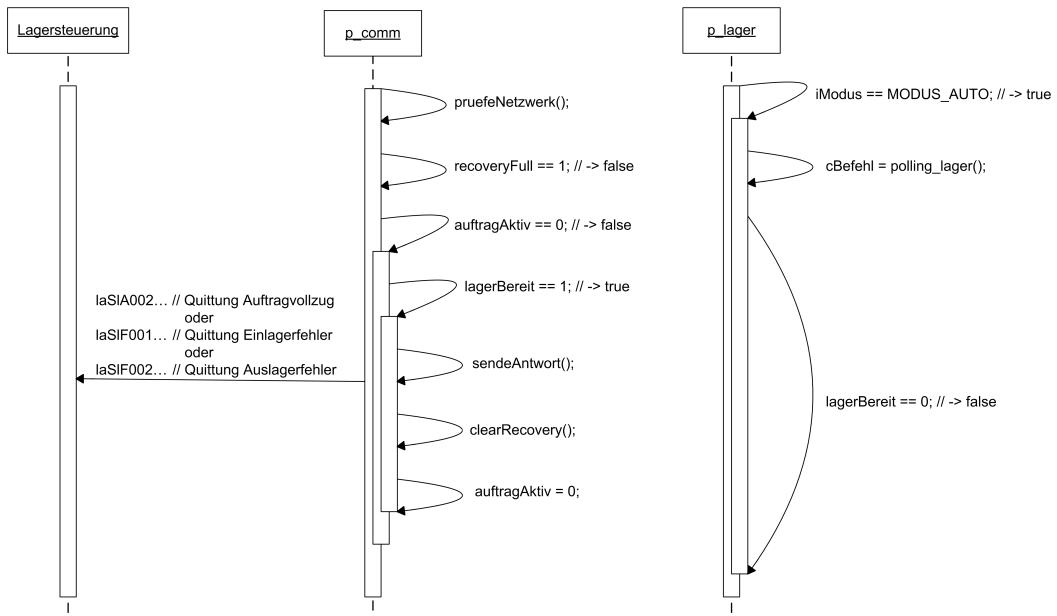
11.1.4.3.2 Auftragsbearbeitung durch Lagerhardware



Wenn Task 1 im MODUS_AUTO ist, wird zunächst wieder der physikalische Zustand des Lagers abgefragt und zwischengespeichert. Da nun durch das Flag lagerBereit erkannt wird, dass die Lagerhardware einen Auftrag zu bearbeiten hat, wird im weiteren Verlauf der zuvor zwischengespeicherte Zustand der Lagerhardware geprüft. Durch die Übergabe und dem Start des Auftrags durch lager() im Task 2 wurde die Hardware aus der Grundstellung in andere Zustände versetzt. Wenn der Zustand der Lagerhardware wieder als in Grundstellung befindlich gemeldet wird, d.h. der Motor arbeitet nicht mehr und die Ventile des Lagers wieder in Grundstellung stehen, wird über buildAnswer() der globale Sendepuffer mit der Quittung zur erfolgreichen Bearbeitung des Auftrags beschrieben. Weiterhin wird das Flag lagerBereit auf eins gesetzt, um die Bearbeitung zum Senden der Quittung in Task 2 fortzuführen. Sollte solange kein Taskwechsel erfolgt, d.h. das Lager arbeitet, wird über den physikalischen Zustand und dem Flag traegerfehler geprüft, ob ein Bestandsfehler aufgetreten ist. Dieses Flag wird durch die neu angebaute Lichtschranke gesetzt, wenn bei einer Einlagerung das Zielfach belegt ist, oder bei einer Auslagerung leer ist. Sollte ein solcher Fehler auftreten, wird der Task sofort gewechselt. Das Lager befindet sich physikalisch in Grundstellung, da das Ventil noch nicht angefahren wurde. Die entsprechende Fehlermeldung wurde bei Auswertung des Auftrags über cbfEinlagern / cfbAuslagern bereits gesetzt.

11.1 Lager

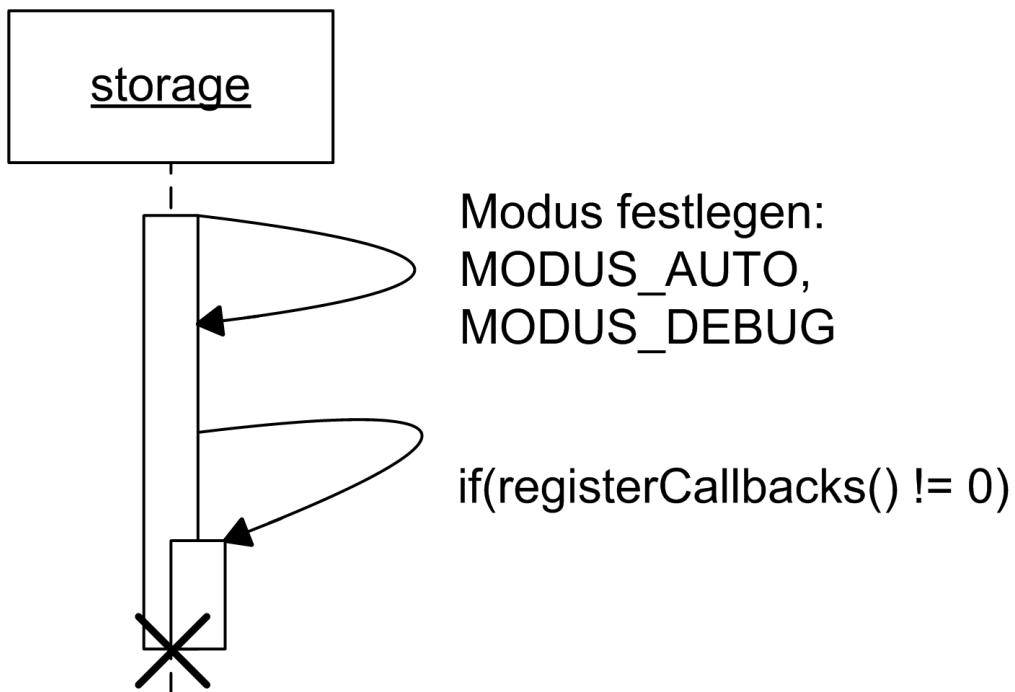
11.1.4.3.3 Auftrag Quittieren



Nach positiver Prüfung des Netzwerks über `pruefeNetzwerk()` und negativer Prüfung des Recovery-Flags `recoveryFull` wird über das Flag `auftragAktiv` festgestellt, dass ein Auftrag in Bearbeitung ist. Da die Antwortnachricht bereits global im Sendepuffer vorbelegt ist, wird dieser nur noch über `sendeAntwort()` an die Lagersteuerung übertragen. Abschließend wird die Recoverydatei geleert und das Flag `auftragAktiv` zurückgesetzt. Das Lager kann nun einen weiteren Auftrag empfangen und bearbeiten. Thread 1 überprüft in dieser Zeit nur seinen Modus und den physikalischen Zustand und wartet wieder auf die Übergabe eines Auftrags.

11.1.4.4 Fehler

11.1.4.4.1 Fehler Callbackregistrierung



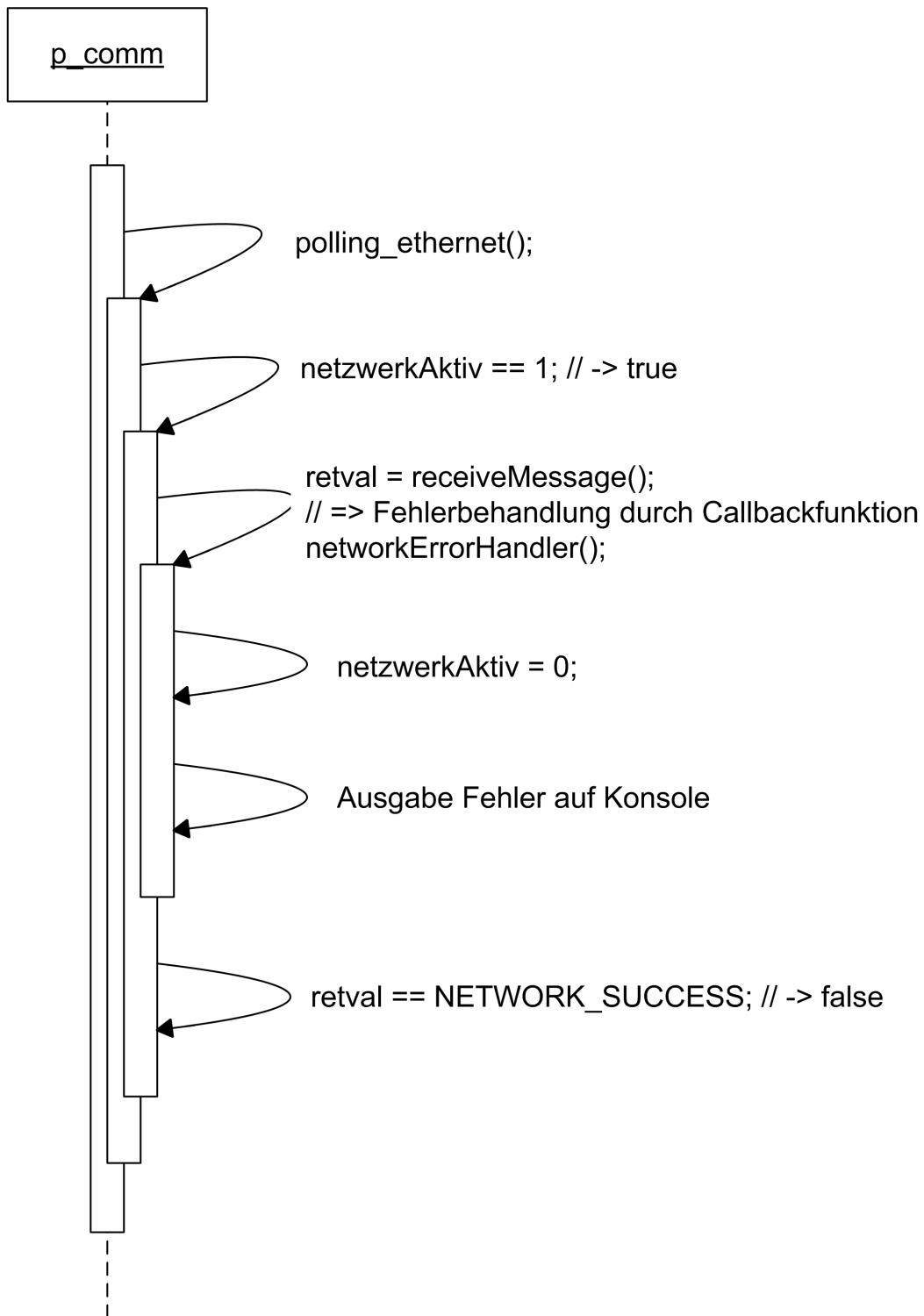
Die Registrierung der Callbackfunktionen `cbfEinlagern()` und `cbfAuslagern()` für das Modul `fdzMessageHandler`, sowie der Fehlerbehandlungsfunktion `networkErrorHandler()` für das Modul `fdzNetwork` findet über die Funktion `registerCallbacks()` beim Hochfahren des Lagers statt. Sollte eines der Standardmodule einen Fehler zurückmelden, werden die Tasks nicht gestartet und das Programm sofort mit einer entsprechenden Fehlermeldung an der Konsole beendet.

11.1.4.4.2 Verbindungsabbruch

Ein Verbindungsabbruch kann an zwei Stellen erkannt werden, beim Empfang einer Nachricht und beim Senden. Die Behandlung eines Netzwerkfehlers findet über eine Funktion immer des Tasks 2 und bei Erkennung eines Fehlers beim Senden statt.

11.1.4.4.2.1 Fehler bei Empfang

11.1 Lager



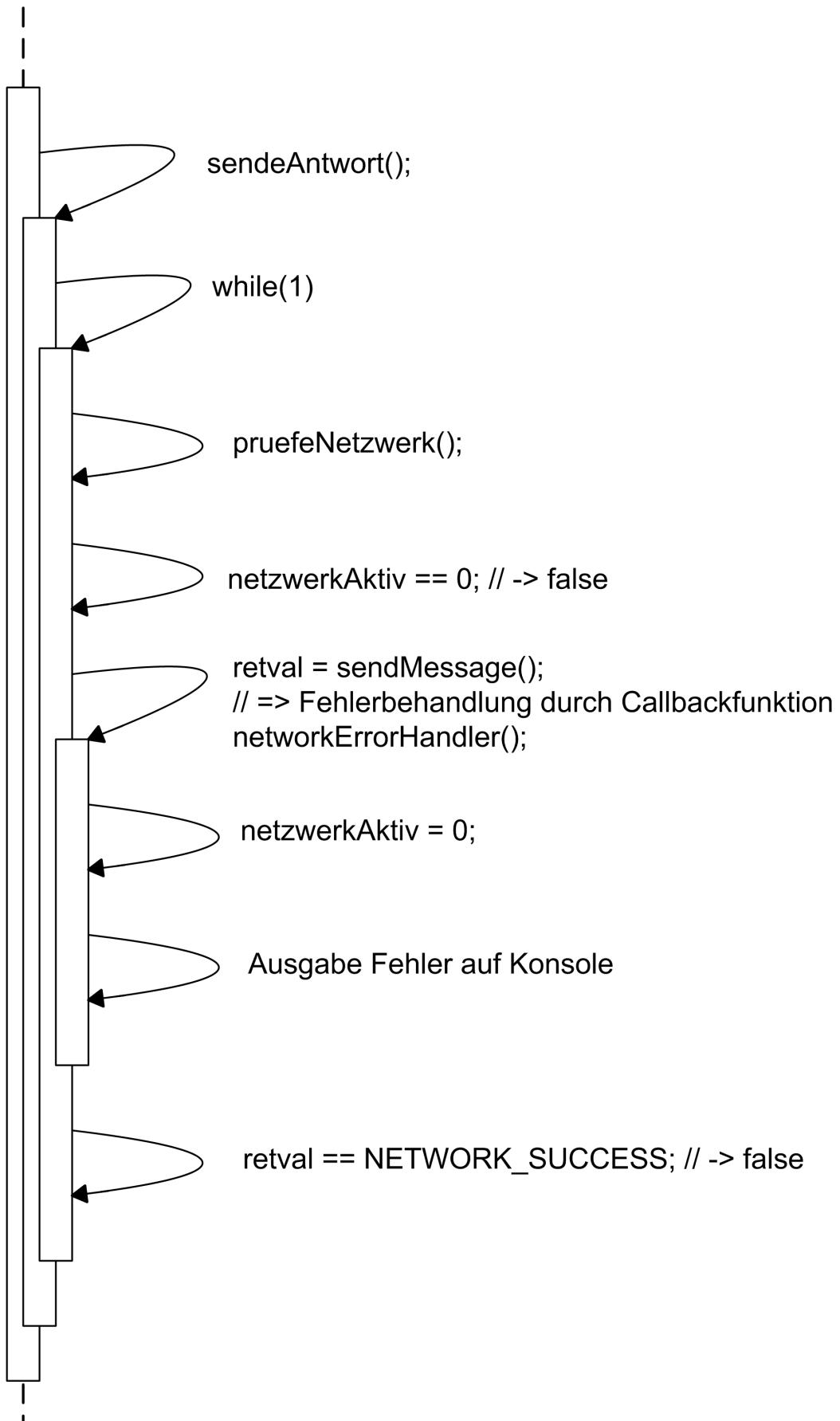
Beim Nachrichtenempfang über `polling_etherne()` wird zunächst durch Prüfung des Flags `netzwerkAktiv` geprüft, ob Nachrichten empfangen werden können. Sollte dies der Fall sein, wird über die API des Moduls `fdzNetwork` der TCP-Stack abgefragt. Wenn innerhalb dieses Moduls ein Fehler erkannt wird, wird die Callbackfunktion `networkErrorHandler()`

11.1 Lager

aufgerufen. Diese setzt das Flag `netzwerkAktiv` auf Null und gibt eine Fehlerbeschreibung an der Konsole aus. Abschließend wird nur noch die Rückgabe des Moduls geprüft, ob eine sinnvolle Tracemeldung des Auftragseingangs auf der Konsole ausgegeben werden soll. Dem ist im Fehlerfall nicht so.

11.1.4.4.2.2 Fehler beim Senden

11.1 Lager

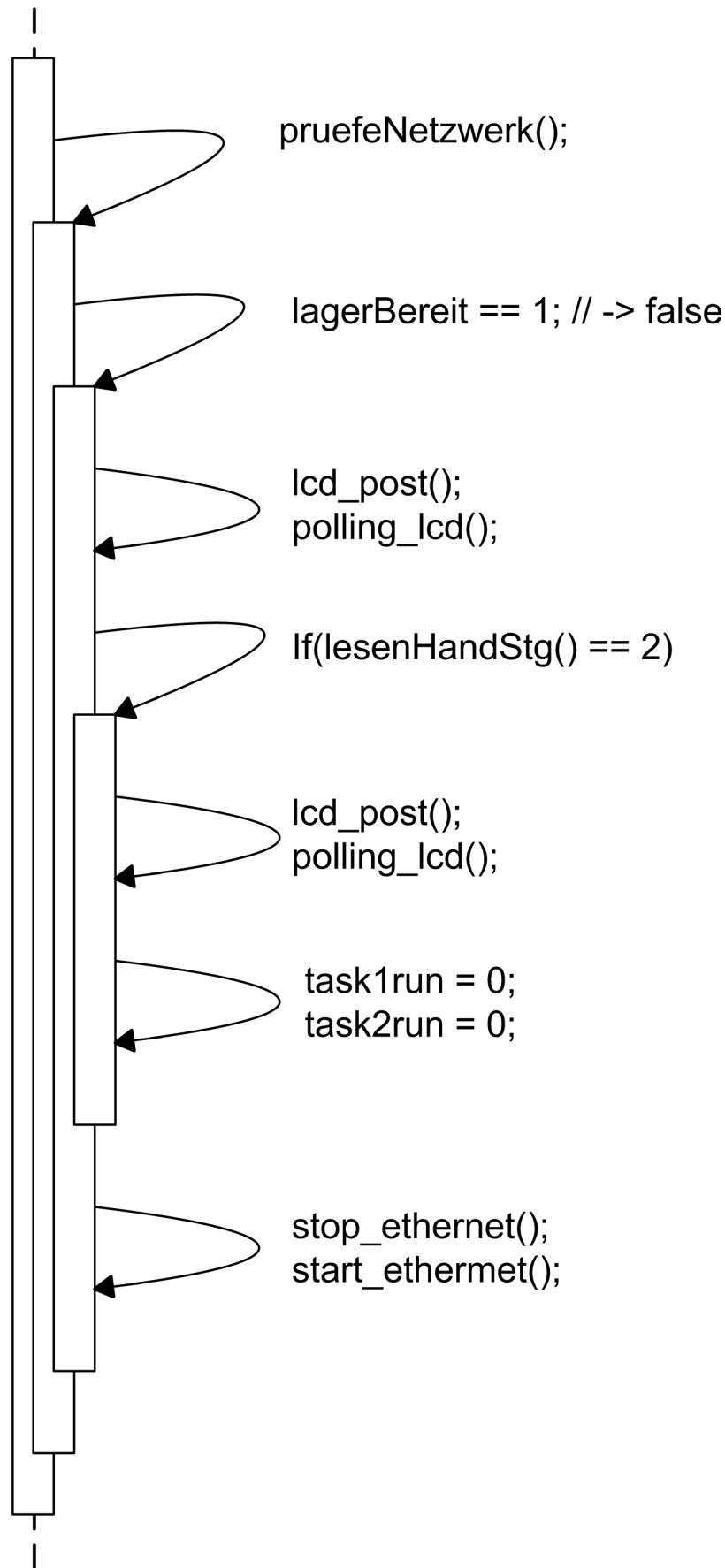


11.1 Lager

Quittungen werden ausschließlich über die Funktion `SendeAntwort()` an die Lagersteuerung geschickt. Diese Funktion kehrt erst zurück, wenn es gelungen ist, die Nachricht zu senden, oder in der Fehlerbehandlungsfunktion `pruefeNetzwerk()` das Lager heruntergefahren wird. In einer While-Schleife wird der Status der Kommunikation über `pruefeNetzwerk()` überprüft. Sollte hier bereits keine aktive Kommunikation existieren, wird dort versucht diese aufzubauen (Siehe Netzwerkfehlerbehandlung Kapitel ??). Anschließend wird über `netzwerkAktiv` geprüft, ob eine Kommunikation möglich ist. Ist dem so, wird über die API des Moduls `fdzNetwork` der global vorbesetzte Nachrichtenpuffer gesendet. Tritt beim Senden innerhalb der API ein Fehler auf, wird die Callbackfunktion `networkErrorHandler()` aufgerufen. Die Abläufe dieser Funktion sind bereits bei Fehler bei Empfang beschrieben. Abschließend wird anhand der Rückgabe des Moduls geprüft, ob eine sinnvolle Tracemeldung des Auftragseingangs auf der Konsole ausgegeben werden soll und die Schleife verlassen werden kann. Dem ist im Fehlerfall nicht so.

11.1.4.4.2.3 Fehlerbehandlung

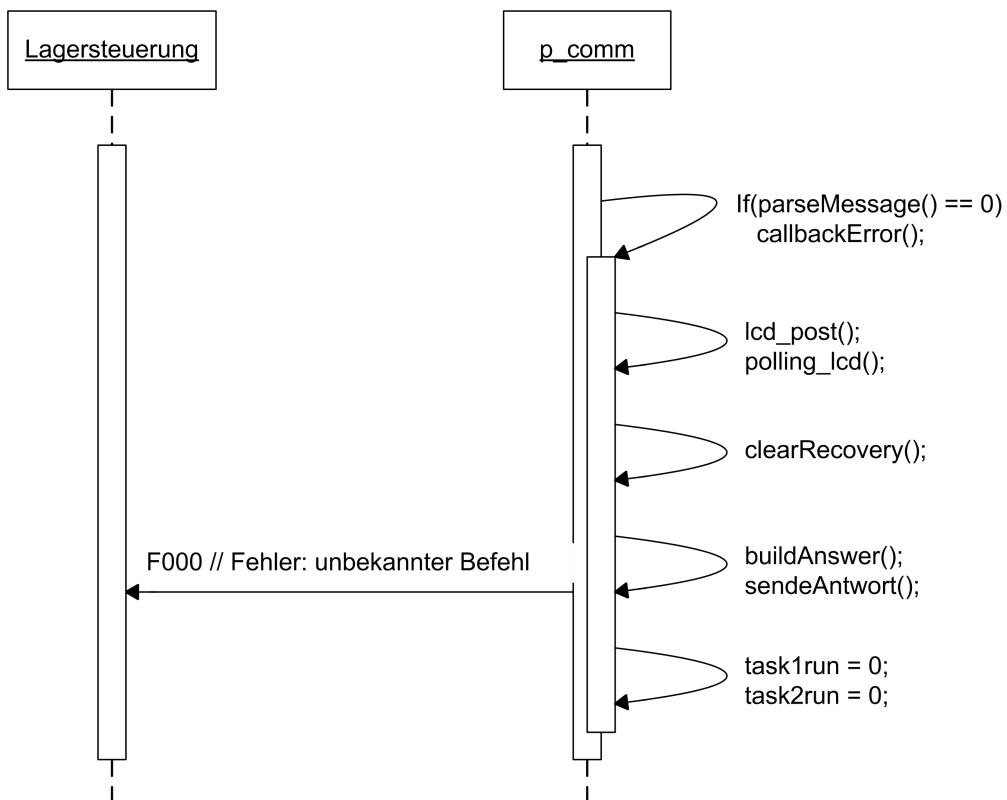
11.1 Lager



11.1 Lager

Die Kontrolle des Kommunikationsstatus sowie die Fehlerbehandlung für Netzwerkfehler finden in der Funktion `pruefeNetzwerk()` in `new_ethernet.c` statt. Zunächst wird über das Flag `lagerBereit` geprüft, ob die Fehlerbehandlung durch den Operator möglich ist, da bei Aktivität der Lagerhardware die Fehlerbehandlung dortiger Fehler Vorrang hat. Nach Setzen und Ausgabe einer Meldung auf dem LCD über `lcd_post()` und `polling_lcd()`, wird die Handsteuerung über `lesenHandStg()` abgefragt. Nur wenn die Taste zwei, d.h. Abbrechen, gedrückt wird, fährt das Lager infolge des Setzens der Flags `task1run` und `task2run` auf null herunter. Das Herunterfahren wird auf dem LCD angezeigt. Sollte zu dem Zeitpunkt keine Taste gedrückt worden sein, wird über `stop_ethernet()` versucht eine eventuell fehlerhafte Verbindung zu beenden, und anschließend mit `start_ethernet()` versucht eine Verbindung zu erstellen. Entsteht eine fehlerfreie Verbindung, so wird das Flag `netzwerkAktiv` in der Funktion auf eins gesetzt.

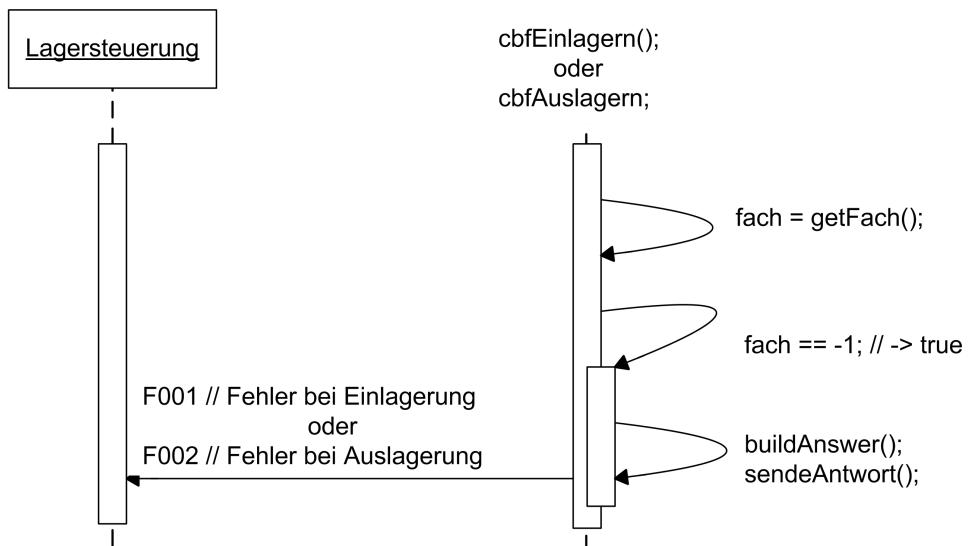
11.1.4.4.3 Auftragskennung unbekannt



Wenn eine Nachricht empfangen und bei deren Analyse über die Funktion `parseMessage()` des Moduls `fdzMessageHandler` keine Callbackfunktion zur Bearbeitung des Kommandos gefunden werden, gibt die Funktion Null zurück. Nach Ausgabe des Fehlers wird die Nachricht aus der Recovery-Datei gelöscht. Weiterhin wird der über `buildAnswer()` die Fehlernachricht `F000` global aufgebaut und über die Funktion `sendeAntwort()` an die Lagersteuerung gesendet. Schließlich wird das Lager durch Setzen der Flags `task1run` und `task2run` auf null heruntergefahren.

11.1 Lager

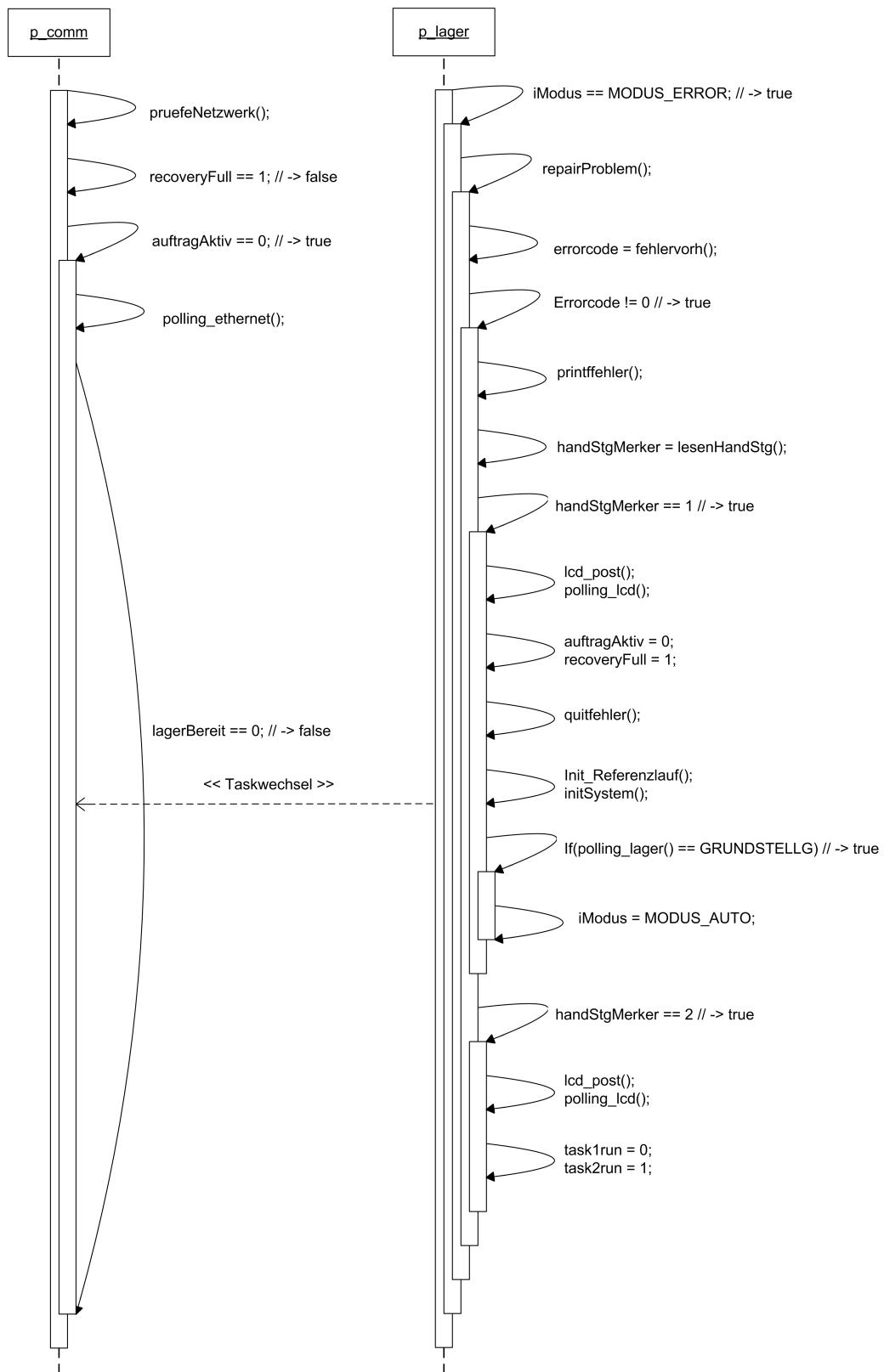
11.1.4.4 logischer Fehler im Auftrag (Fach falsch)



Vor Sicherung des Auftrags in die Recovery-Datei und Übergabe des Auftrags an die Lagersteuerung wird in den Callbackfunktionen `cbfEinlagern()` und `cbfAuslagern()` das Fach aus dem Nachrichtenstring zu einer Zahl geparsst. Dabei wird auch geprüft, ob das angegebene Fach zur Ein- bzw. Auslagerung gültig ist (1-15). Ist das Fach ungültig, wird die dem Auftrag entsprechende Fehlermeldung über `buildAnswer()` global aufgebaut und über `sendeAntwort()` an die Lagersteuerung übertragen.

11.1 Lager

11.1.4.4.5 Fehler Lagerhardware (Notaus, etc.)



11.1 Lager

Wenn innerhalb der Lagerhardware ein Fehler festgestellt wird, wechselt der Modus der Hardwaresteuering zu MODUS_ERROR. Der aufgetretene Fehler wird in einer Fehlerqueue in `error.c` gespeichert. Die Vorgänge, wie die Umschaltung erfolgt sind ausschließlich aus dem alten Softwarestand übernommen, und werden hier nicht weiter behandelt. Im MODUS_ERROR wird die Fehlerbehandlungsfunktion `repairProblem()` aufgerufen. Die Funktion überprüft zunächst, ob eine gültige Fehlernummer existiert und gibt diese zurück. Über die Funktion `printffehler()` wird ein der Fehlernummer entsprechende Fehlermeldung mit Wahlmöglichkeit, den Auftrag zu wiederholen oder abzubrechen auf dem LCD ausgegeben. Danach wird das Flag `auftragAktiv` auf Null zurückgesetzt und das Flag `recoveryFull` auf eins gesetzt, um im Falle einer Wiederholung auf dem Recovery aufsetzen zu können. Über die Funktion `lesenHandStg()` wird die Handsteuerung abgefragt. Wurde die Taste eins zum Wiederholen des Auftrags gedrückt, wird das LCD entsprechend aktualisiert, über `quitfehler()` der Fehler gelöscht und versucht die Lagerhardware über die Funktionen `Init_Refenzlauf()` und `initSystem()` in Grundstellung zu fahren. Bei erfolgreicher Beendigung der Grundstellungsfahrt wird automatisch in `initSystem()` das Flag `lagerBereit` zum Taskwechsel gesetzt. Der Taskwechsel findet erst nach Verlassen von `repairProblem()` statt. Über `polling_lager()` wird anschließend geprüft, ob das Lager tatsächlich die Grundstellung erreicht hat. Nur wenn das der Fall ist, wird der Modus der Hardwaresteuering wieder in den MODUS_AUTO versetzt. Durch Setzen der Flags setzt sich die Bearbeitung wie die Bearbeitung eines Recovery-Falles fort. Die entsprechende Beschreibung befindet sich im Kapitel ??.

11.1.4.4.6 Fehler Recovery-Bearbeitung

Die Recovery-Bearbeitung wird über die Funktionen `readRecovery()`, `saveRecovery()`, `clearRecovery()` und `isRecoveryEmpty()` in der Datei `recover.c` behandelt. Die Aufrufslogik wurde bereits bei der Beschreibung einer Auftragsbearbeitung abgebildet. Ein Fehler bei der Recovery-Bearbeitung bedeutet ein Lese-/Schreibfehler auf den Datenträger des Lagerchips. In diesem Fall wird das Lager ohne Rückmeldung eines eventuell aktiven Auftrags sofort durch das Setzen der Flags `task1run` und `task2run` auf Null heruntergefahren, da eine stabile Auftragsbearbeitung nicht mehr möglich ist.

11.1.4.5 Bekannte Fehler und Verbesserungsvorschläge

- Fehler sehr selten: erst Ventil, dann Motor angefahren
(Vermutung Ursache: Motortimeout in ungünstigem Moment)

11.1.4.6 Erweiterung der Lichtschrankenprüfung

- Prüfung am Ende einer Einlagerung, ob Palette tatsächlich im Fach steht
- Prüfung bei Auslagerung, vor Querverschiebung der Palette auf den Transportschlitten, ob die Palette noch vorhanden ist.

11.2 Lagersteuerung

11.2.1 Beschreibung der Bestandverwaltung

Die zentrale Klasse der Lagersteuerung ist die Klasse Storage. Sie verwaltet den Bestand der Smarties, überprüft den Bestand auf Konsistenz, wählt die für die Bearbeitung nötigen Fächer aus, etc. Sie ist dabei auf die Klassen ColorMatrix, Pallet und Inventory angewiesen, die zuerst erklärt werden.

11.2.1.1 Beschreibung von ColorMatrix

Die Klasse ColorMatrix verwaltet Anzahl, Position und Farbe der Smarties, das Klassendiagramm unter ?? verdeutlicht den Aufbau.

ColorMatrix	
- matrix	: string
- pallet_size	: unsigned int
+ ColorMatrix(: int)
+ setMatrix(: const string&) : int
+ getMatrix()	: string
+ hasColor()	: char
+ countNumberOf(: char) : int
+ countSmarties()	: int
+ getPalletSize()	: int

Abbildung 11.1: Klassendiagramm der Klasse ColorMatrix

11.2.1.1.1 Der Konstruktor

Um die Wiederverwertbarkeit der Klasse zu erhöhen wird per Konstruktor die Größe der matrix festgelegt. Dies muss erfolgen, da jede Zuweisung einer Matrix einer anderen Größe eine Exception wirft. Die festgelegte Größe kann nicht mehr geändert, durch getPalletSize() aber erfragt werden.

11.2.1.1.2 get / set Matrix

Die Matrix muss entweder die Länge, die im Konstruktor gesetzt wurde, oder die Länge eins und den Inhalt - , haben. Der Rückgabewert von setMatrix(**const** string&) ist die Anzahl der

11.2 Lagersteuerung

Smarties einer Farbe, kommen mehrere Farben, gar keine, oder eine Matrix falscher Größe vor, wird eine Exception geworfen. `getMatrix()` gibt die Matrix als String zurück.

11.2.1.1.3 Informationen über die Matrix

Die Methode `hasColor()` gibt die erste in der Matrix gefundene Farbe als char zurück. `countSmarties()` gibt die Anzahl der Smarties als int zurück. `countNumberOf(char)` gibt die Anzahl der Smarties der Farbe char als Alle 3 Methoden werfen bei fehlerhafter Matrix Exceptions, was in der Praxis nicht vorkommt, da Fehler bereits bei `ColorMatrix::setMatrix()` erkannt werden.

11.2.1.2 Beschreibung von Pallet

Die Klasse Pallet verwaltet Position, Name und Einlagerdatum der Palette. Sie hat die unter ?? beschriebene ColorMatrix als Membervariable und reicht alle Aufrufe zur Verwaltung der Matrix zu ihr durch. Das Klassendiagramm unter ?? verdeutlicht den Aufbau.

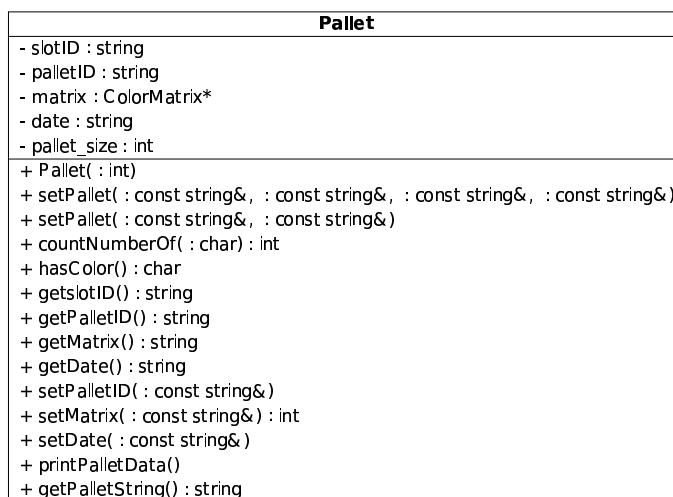


Abbildung 11.2: Klassendiagramm der Klasse Pallet

11.2.1.2.1 Der Konstruktor

Um die Wiederverwertbarkeit der Klasse zu erhöhen wird per Konstruktor die Größe der Matrix festgelegt. Dies muss erfolgen, da jede Zuweisung einer Matrix einer anderen Größe eine Exception wirft. Die festgelegte Größe kann nicht mehr geändert werden.

11.2.1.2.2 setPallet

Zum Setzen der Pallettendaten gibt es mehrere Möglichkeiten:

11.2 Lagersteuerung

```
setPallet(const string& Fachnummer, const string& Palletten-ID, const
          string& Matrix, const string& Einlagerdatum)

setPallet(const string& Fachnummer, "-")
```

Wobei die zweite Methode nur für leere Fächer geeignet ist.

Des weiteren sind alle Attribute über setMethoden zu erreichen. (siehe ??)

11.2.1.3 Beschreibung von Inventory

Die Klasse Inventory übernimmt das Einlesen der Bestandsdatei. Die Überprüfung, ob der Inhalt korrekt ist findet erst in der Klasse Storage statt. Da die Methodennamen selbsterklärend sind wird im folgenden nur der Konstruktor erklärt und für weiteres auf ?? verwießen.

Inventory	
-	line : vector< string >
-	pallet_size : int
-	readBDatei(: const string&) : int
+	Inventory(: int, : const string&)
+	getName() : string
+	getSlotQuantity() : int
+	getFreeSlots() : int
+	getNumberOf(: char) : int
+	getSlotNumber(: unsigned int) : string
+	getPalletName(: unsigned int) : string
+	getMatrix(: unsigned int) : string
+	getDate(: unsigned int) : string

Abbildung 11.3: Klassendiagramm der Klasse Inventory

11.2.1.3.1 Der Konstruktor

Um die Wiederverwertbarkeit der Klasse zu erhöhen wird per Konstruktor die Größe der Matrix und die einzulesende Bestandsdatei festgelegt. Es wird automatisch die Bestandsdatei eingelesen und zeilenweise in den vector line geschrieben. Anschließend kann über die übrigen Methoden Informationen aus der Bestandsdatei gelesen werden.

11.2.1.4 Beschreibung von Storage

Die Klasse Storage enthält alle Funktionen zur Verwaltung des Lagers. Das Lager besteht aus einem Palettenlager mit mehreren Fächern, die entweder leer sind, oder mit mindestens einem

11.2 Lagersteuerung

Smartie bestückte Paletten enthalten. Die Größe des Lagers und der Paletten ist variabel, muss aber bereits bei der Definition des Objektes festgelegt werden und kann anschließend nicht mehr verändert werden. Die Bestandsdaten werden von der Klasse Inventory gelesen und das Objekt der Klasse Storage automatisch damit initialisiert.

Für den jetzigen Ausbau kommt ein Palettenlager mit 15 Fächern, und maximal 91 Smarties pro Palette zum Einsatz. Es wurde darauf geachtet, dass auch Palettenlager mit anderen Dimensionen verwaltet werden können.



Abbildung 11.4: Klassendiagramm der Klasse Storage

11.2.1.4.1 Der Konstruktor

Dem Konstruktor von Storage müssen 3 Argumente übergeben werden.

1. Die Anzahl der Lagerfächer
2. Die Größe der Lagerpaletten
3. Der Name der Bestandsdatei

`Storage(int, int, const string&)`

Der Konstruktor erstellt ein Objekt der Klasse Inventory und initialisiert in der Funktion `readInventory(int, int, const string&)` das Storage Objekt mit den Werten aus der

11.2 Lagersteuerung

Bestandsdatei. Anschließend überprüfen die Funktionen `uniquePallets()`, `uniqueSlots()`, `countSlots()` und `countFreeSlots()` ob der Inhalt der Bestandsdatei syntaktisch und, soweit dies möglich ist, inhaltlich korrekt ist. Tritt ein Fehler auf, so wird eine Exception geworfen und es wird kein Objekt erstellt.

11.2.1.4.2 letsDoStorage

Müssen Befehle an das Lager gesendet werden, so wird der Befehl an

```
letsDoStorage(const string& aCommand, int aSlotNumber)
```

übergeben. Diese Funktion veranlasst das Senden und Empfangen der Nachrichten an und vom Lager.

11.2.1.4.3 writeInventory

Hängt an den Dateinamen der alten Bestandsdatei die Endung .bak an und schreibt die aktualisierten Daten unter den ursprünglichen Namen der Bestandsdatei. Die zu schreibenden Daten werden über `getStorageString()` abgefragt.

11.2.1.4.4 chooseSlotToTakeOut

Wählt die Palette zum Auslagern aus. Es wurde nicht der unter ?? beschriebene Algorithmus implementiert, sondern es wird die erste Palette ausgewählt, welche die entsprechende Farbe enthält.

11.2.1.4.5 getFreeSlotNumber

Wählt ein freies Fach zum Einlagern einer Lagerpalette aus.

11.2.2 Beschreibung der Abläufe

11.2.2.1 Gesamtablauf

Nach dem Starten der Lagersteuerung wird in der main-Funktion als erstes ein Storage-Objekt erzeugt. (siehe ??) Als nächstes ließt die Funktion `recoverSystem()` die temporäre Log-Datei ein und überprüft anhand der eingelesenen Informationen, ob die Lagersteuerung im Recover-Modus gestartet werden muss (siehe Kapitel ??). Anschließend werden die Callbackfunktionen für die Befehle der Steuerung registriert (siehe Kapitel ??), danach die Netzwerkverbindungen zum Lager und zur Steuerung hergestellt (siehe Kapitel ??) und als letztes durch Aufrufen der Funktion `awaitCommand()` in die Empfangsschleife zum Empfangen von Befehlen von der

11.2 Lagersteuerung

Steuerung gesprungen. Sobald eine in der Funktion `registerCallbacksControl()` registrierte Nachricht von der Steuerung empfangen wird, wird einer der in den folgenden Kapiteln beschriebenen Vorgänge durch Aufrufen der entsprechenden Callback-Funktion gestartet.

11.2.2.2 Allgemeiner Ablauf für die Abarbeitung eines Kommandos

Das nachfolgend beschriebene Grundgerüst gilt für alle Callback-Funktionen der Lagersteuerung. Zunächst wird der Zähler für den Abarbeitungs-Status (globale Variable `recoveryStatus`) um 1 erhöht, da mit dem Aufrufen der entsprechenden Callback-Funktion ein gültiger Befehl von der Steuerung empfangen wurde. Der Recovery-Status, sowie der empfangene Befehl werden mit der Helper-Funktion `logS()`, bzw der Funktion `writeLog()` der Klasse Logging in die temporäre Log-Datei geschrieben.

Um auf das Kommando der Steuerung korrekt antworten zu können, wird als nächstes mit der Funktion `extractMessageID()` die Message-ID aus der empfangenen Nachricht herausgelesen und in der globalen Variable `messageID` gespeichert. Anschließend wird durch Aufrufen der Funktion `send_A001()` die Bestätigung für das Erhalten des Kommandos an die Steuerung gesendet.

Als nächstes wird überprüft, ob beim Starten der Lagersteuerung während des Einlesens der Bestandsdatei ein Fehler aufgetreten ist. In diesem Fall gäbe es kein Storage-Objekt und die globale Variable `theStorage` würde einen NULL-Pointer enthalten. Ist dies der Fall, wird durch Aufruf der Funktion `buildMessage()` die Fehlernachricht für F001 generiert und der Funktion `send_A002()` zum senden an die Steuerung übergeben.

Wurde die Bestandsdatei korrekt eingelesen, wird anstelle einer Fehlernachricht zu senden der individuelle Ablauf des jeweiligen Vorgangs gestartet und je nach dem, ob der Vorgang ohne Fehler ausgeführt werden konnte, oder nicht, anschließend eine Bestätigung oder eine entsprechende Fehlernachricht an die Steuerung gesendet. Die individuellen Abläufe der jeweiligen Vorgänge werden in den folgenden Kapiteln beschrieben.

Die Funktionen `send_A001()` und `send_A002()` erhöhen jeweils den Zähler `recoveryStatus` und schreiben die Informationen in die temporäre Log-Datei.

Nach dem Senden der zweiten Nachricht ist der Vorgang abgeschlossen und der Schrittzähler für das Recovery wird wieder auf 0 zurückgesetzt. Anschließend wartet die Lagersteuerung in der Empfangsschleife `awaitCommand` wieder auf Kommandos der Steuerung.

11.2.2.3 Individueller Ablauf für die Abarbeitung der einzelnen Kommandos

11.2.2.3.1 Bestandsanfrage

Callback-Funktion: `callback_quantitySmarties()`

Bei der Bestandsanfrage für eine Farbe wird die Methode `quantitySmarties()` der Klasse `Storage` aufgerufen. Sie liefert die Anzahl der sich im Lager befindlichen Smarties einer angefragten Farbe zurück. Wird der Funktion eine Falsche Farbe übergeben wirft sie die Exception `ERR_INVALID_COLOR`.

11.2 Lagersteuerung

11.2.2.3.2 Anfrage auf Verfügbarkeit von Smarties

Callback-Funktion: `callback_smartiesAvailable()`

Bei der Anfrage auf Verfügbarkeit von Smarties einer bestimmten Farbe wird die Methode `smartiesAvailable()` der Klasse Storage aufgerufen. Anhand des Rückgabeparameters überprüft die Callback-Funktion, ob der Bestand ausreicht. Ist dies nicht der Fall, wird die Methode `quantitySmarties()` der Klasse Storage aufgerufen, welche die Anzahl der Smarties der Farbe zurückliefert. Anschließend wird damit die Antwort-Nachricht für die Steuerung generiert.

11.2.2.3.3 Anfrage auf freie Stellplätze

Callback-Funktion: `callback_slotsAvailable()`

Diese Callback-Funktion ruft `getSlotsAvailable()` der Klasse Storage auf, welche die Anzahl der freien Stellplätze zurückliefert. Anschließend wird mit dem Rückgabewert die Antwort-Nachricht für die Steuerung generiert.

11.2.2.3.4 Einlagern

Callback-Funktion: `callback_putIn()`

Der eigentliche Einlagervorgang wird durch Aufrufen der Methode `putIn()` der Klasse Storage gestartet. Diese überprüft, ob die übergebene Matrix korrekt ist, ob ein freies Lagerfach vorhanden ist, und ob sich die übergebene Palletten-ID nicht bereits im Lager befindet. Im Fehlerfall wird eine entsprechende Exception geworfen. Im Gutfall wird das Einlagerkommando für das Lager generiert und die Funktion `letsDoStorage()` aufgerufen, welche das Kommando an das Lager sendet und die Antworten empfängt und auswertet. Die Methode `putIn()` ist des Weiteren dafür verantwortlich, die Bestandsdaten durch Aufrufen von `updateInventoryPutIn()` und `writeInventory()` zu aktualisieren.

Die Callback-Funktion fängt geworfene Exceptions und generiert entsprechende Antworten für die Steuerung.

11.2 Lagersteuerung

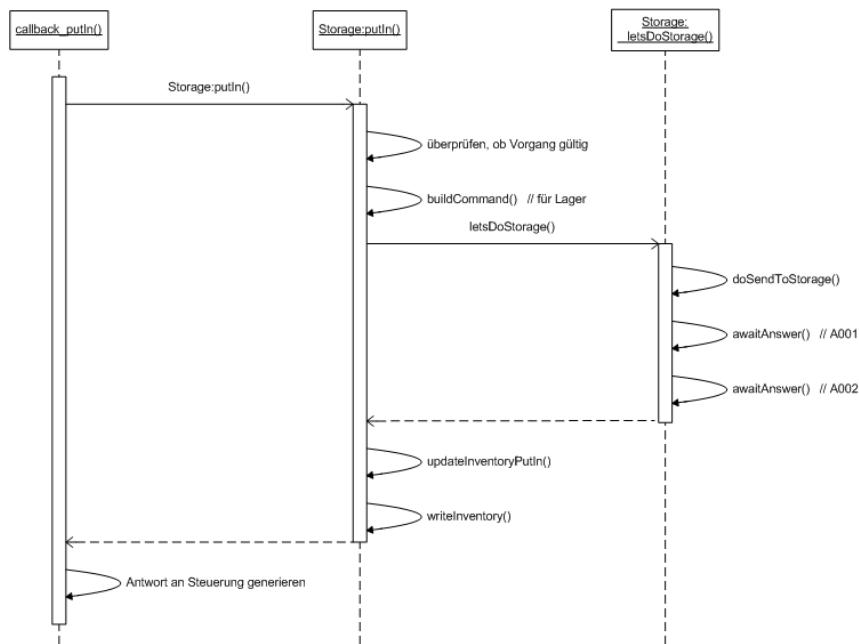


Abbildung 11.5: Einlagrervorgang

11.2.2.3.5 Auslagern

Callback-Funktion: `callback_takeOut()`

Der eigentliche Auslagrervorgang wird durch Aufrufen der Methode `takeOut()` der Klasse `Storage` gestartet. Diese überprüft, ob sich genügend Smarties im Lager befinden und wählt durch Aufrufen der Funktion `chooseSlotToTakeOut()` das auszulagernde Fach aus. Im Fehlerfall wird eine entsprechende Exception geworfen. Im Gutfall wird das Auslagerkommando für das Lager generiert und die Funktion `letsDoStorage()` aufgerufen, welche das Kommando an das Lager sendet und die Antworten empfängt und auswertet. Die Methode `takeOut()` ist des Weiteren dafür verantwortlich, die Bestandsdaten durch Aufrufen von `updateInventoryTakeOut()` und `writeInventory()` zu aktualisieren.

Die Callback-Funktion fängt geworfene Exceptions und generiert entsprechende Antworten für die Steuerung.

11.2 Lagersteuerung

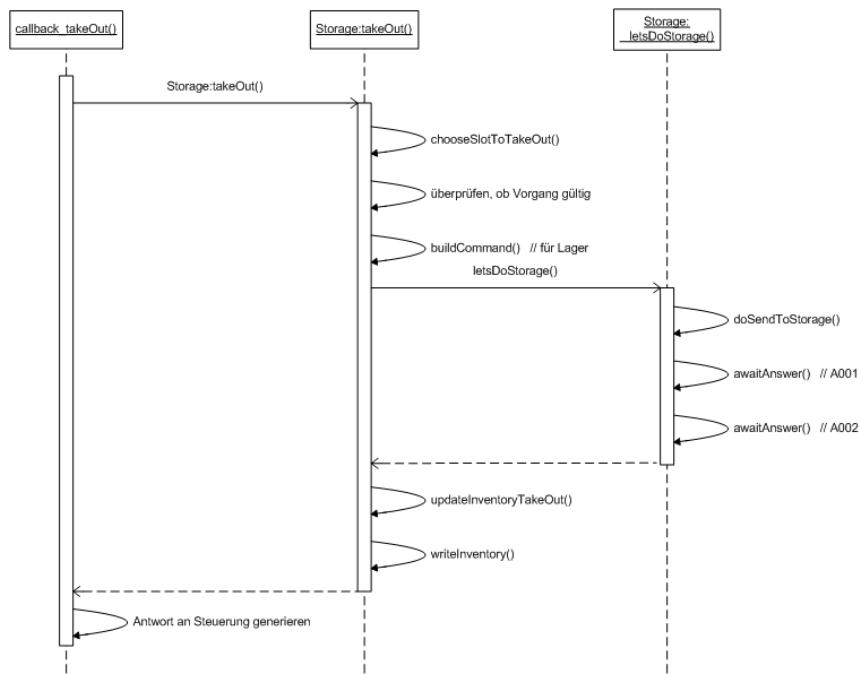


Abbildung 11.6: Auslagervorgang

11.2.2.3.6 Herunterfahren

Callback-Funktion: `callback_shutdown()`

Die Callback-Funktion zum Herunterfahren der Lagersteuerung unterscheidet sich vom Ablauf der anderen Vorgänge dadurch, dass vor dem Senden des A002 keine Überprüfung stattfindet, ob die Bestandsdatei vorher korrekt eingelesen werden konnte (globale Variable `theStorage == NULL`). Deswegen wird auch beim Einlesen einer fehlerhaften Bestandsdatei kein F001 gesendet, sondern ein A002 für die Bestätigung des Herunterfahrens. Anschließend werden die Netzwerkverbindungen zum Lager und zur Steuerung durch Aufrufen der Funktionen `closeSocket()` und `cleanup()` des fdz-Netzwerkmoduls geschlossen. Danach beendet sich das Programm.

11.2.3 Beschreibung der Callback-Funktionen

Wird eine Callback-Funktion aufgerufen, so wird das Abarbeiten eines Vorgangs, z.B. das Ein- oder Auslagern einer Palette, gestartet. Die Callback-Funktionen werden aufgerufen, sobald die Funktion `parseMessage()` aufgerufen wird, und dieser eine Adresse vom Typ SMessage übergeben wird, wobei SMessage mit einer zuvor registrierten Nachricht initialisiert wurde.

Die Funktion `parseMessage()` wird in den Netzwerk-Funktionen `awaitCommand()` (bei empfangenen Kommandos von der Steuerung), `awaitAnswer()` (bei empfangenen Antworten vom Lager) und `recoverSystem()` (für den Recovery-Vorgang) aufgerufen.

Beispiel:

11.2.3.1 Beispielcode für Callbacks

Listing 11.1: Beispielcode für Callbacks

```
// Nachricht registrieren, bei der Callback-Funktion aufgerufen werden soll
SMessage m;
initMessage(&m, "Nachricht vom Netzwerk");
SCallbackData scd;
scd.message = &m;
registerCallback(aufzurufende_Callbackfunktion(), &scd);

// Nachricht initialisieren
SMessage testCallback;
initMessage(&testCallback, "z.B. Nachricht vom Netzwerk");

//Callback-Funktion aufrufen
parseMessage(&testCallback);
```

Das Registrieren der Nachrichten für die Lagersteuerung übernehmen die Funktionen `registerCallbacksControl()`, welche in der Main-Funktion aufgerufen wird und `registerCallbacksStorage()`, welche im Konstruktor der Klasse Storage aufgerufen wird. Die Registrierungs-Funktionen befinden sich in den Quelltextdateien `CallbackFunktionsControl.cpp` und `CallbackFunktionsStorage.cpp`. Die aufrufbaren Callback-Funktionen befinden sich ebenfalls in den entsprechenden Dateien.

Die Funktionen `initMessage()` und `parseMessage()` befinden sich in den Modulen `fdzMessgeHandler` und `fdzNetwork`.

Für jeden Vorgang, den die Lagersteuerung durchführen können soll, muss eine entsprechende Nachricht, sowie Callback-Funktion registriert werden. In der jetzigen Version der Lagersteuerung gibt es Callback-Funktionen für folgende Vorgänge:

- Bestandsanfrage
- Anfrage auf Verfügbarkeit einer Farbe
- Anfrage auf freie Stellplätze
- Einlagern
- Auslagern
- Herunterfahren

Sobald über das Netzwerk eine für einen Vorgang registrierte Nachricht eintrifft, wird der entsprechende Vorgang gestartet, indem die jeweilige Callback-Funktion durch `parseMessage()` aufgerufen wird.

11.2.4 Beschreibung der Netzwerkkommunikation

Die Funktionen, die zum Verbindungsaufbau und zum Senden und Empfangen von Nachrichten der Lagersteuerung benötigt werden, befinden sich in der Datei Network.cpp. Die entsprechenden Funktionen rufen wiederum Funktionen des fdz-Netzwerk-Moduls auf.

11.2.4.1 Aufbau der Netzwerkverbindungen

Beim Starten der Lagersteuerung werden zunächst die Netzwerkverbindungen zum Lager und zur Steuerung hergestellt. Dazu werden in der Main-Funktion, bzw. in der Funktion `recoverSystem()` die Funktionen `ConnectToStorage()` und `ConnectToControl()` aufgerufen. Die Kommunikations- und Verbindungs-IDs werden in folgenden globalen Variablen gespeichert:

- `connID_Control` (Kommunikations-ID für Steuerung)
- `connID_Storage` (Kommunikations-ID für Lager)
- `connID_StorageConnect` (Verbindungs-ID für Lager)

Diese IDs werden zum Senden und Empfangen von Nachrichten benötigt.

11.2.4.2 Empfangen von Nachrichten

Zum Empfangen von Kommandos von der Steuerung wird in der Main-Funktion, bzw. in der Funktion `recoverSystem()` nach dem Starten der Lagersteuerung die Funktion `awaitCommand()` aufgerufen. Sie ruft die Funktion `receiveMessage()` des fdz-Netzwerk-Moduls auf. Diese blockiert das Programm solange, bis entweder eine Nachricht über das Netzwerk eintrifft, oder die Verbindung zur Steuerung unterbrochen wurde. Wird eine Nachricht empfangen, so wird die Message-ID der Nachricht in einer globalen Variable gespeichert. Der Aufruf von `extractMessageID()` findet in der jeweiligen Callback-Funktion statt.

Zum Empfangen von Nachrichten vom Lager wird in der Funktion `letsDoStorage()` der Klasse Storage bei einem Ein- bzw. Auslagervorgang die Funktion `awaitAnswer()` aufgerufen. Diese ruft wiederum die Funktion `receiveMessage()` des fdz-Netzwerk-Moduls auf.

Die Funktionen `awaitCommand()` und `awaitAnswer()` erkennen und reagieren jeweils auf Netzwerkfehler.

Nachdem eine Nachricht von der Steuerung oder dem Lager empfangen wurde, wird die Funktion `parseMessage()` des fdzMessage-Handlers aufgerufen, welche dann die entsprechende Callback-Funktion aufruft (siehe Kapitel ??: Beschreibung der Callbacks)

Eine Überprüfung, ob eine Nachricht aufgrund des Recovery-Status empfangen werden muss, das Erhöhen des Recovery-Schrittzählers, sowie das Schreiben der empfangenen Nachrichten in die temporäre Log-Datei übernehmen jeweils die aufrufenden Funktionen.

11.2 Lagersteuerung

11.2.4.3 Senden von Nachrichten

Beim Senden von Antworten an die Steuerung werden die Funktionen `send_A001()`, bzw. `send_A002()` verwendet. Sie überprüfen zunächst, ob ein Senden der Nachricht aufgrund des Recovery-Status durchgeführt werden darf. Wenn ja, wird die gesendete Nachricht an die temporäre Log-Datei angehängt und der Recovery-Schrittzhäler um 1 erhöht.

Zum endgültigen Senden wird die Funktion `doSendToControl()` aufgerufen. Sie erkennt und reagiert schließlich auf Netzwerkfehler.

Das Senden von Kommandos an das Lager und das Erkennen von Verbindungsfehlern übernimmt die Funktion `doSendToStorage()`. Beim Senden einer Nachricht an das Lager muss zusätzlich eine eindeutige Message-ID generiert werden. Dies übernehmen bereits die Funktionen `putIn()` und `takeOut()` der Klasse Storage durch Aufruf von `buildCommand()`.

11.2.4.4 Schließen der Netzwerkverbindungen

Bei Beenden des Programms (Steuerung sendet K006 - Aufruf von `callback_shutdown()`) werden die offenen Netzwerkverbindungen zur Steuerung und zum Lager geschlossen. Dazu werden die Funktionen `closeSocket()` und `cleanup()` des fdzNetzwerk-Moduls aufgerufen.

Tritt beim Senden, bzw. Empfangen einer Nachricht ein Verbindungsfehler auf, wird vor einem erneuten Verbindungsaufbau ebenfalls `closeSocket()` für die fehlerhafte Verbindung aufgerufen.

11.2.4.5 Verhalten bei Netzwerkfehlern

Die Funktionen `awaitCommand()` - zum Empfangen von Kommandos von der Steuerung, `awaitAnswer()` - zum Empfangen von Nachrichten vom Lager, `doSendToControl()` und `doSendToStorage()` erkennen und reagieren jeweils auf Netzwerkfehler. Sobald in den genannten Funktionen `receiveMessage()`, bzw. `sendMessage()` des fdz-Netzwerk-Moduls aufgerufen werden und diese einen NETWORK_ERROR zurückliefern, wird zunächst versucht, die fehlerhafte Verbindung zu schließen. Anschließend erfolgt ein neuer Verbindungsaufbau durch Aufruf von `connectToControl()`, bzw. `connectToStorage()`, die das Programm so lange blockieren, bis die Verbindung wieder hergestellt wurde.

Ging die Verbindung vor dem Senden einer Nachricht verloren, so wird zunächst so lange versucht, die Verbindung wieder aufzubauen, bis sie wieder hergestellt werden konnte. Anschließend wird die zu sendende Nachricht auf das Netzwerk geschickt.

Ging die Verbindung während des Wartens auf eine Nachricht verloren, so erfolgt ebenfalls ein Reconnect. Anschließend wartet die Lagersteuerung weiter auf eingehende Nachrichten.

11.2.5 Beschreibung des Recovery

11.2.5.1 Zerlegen der Vorgänge in Einzelschritte

Die Recoverylogik ist dafür verantwortlich, dass nach einem Neustart der Lagersteuerung ein zuvor nicht vollständig abgeschlossener Vorgang an der entsprechenden Stelle fortgesetzt wird. Um dies zu ermöglichen, wird jeder Vorgang in Einzelschritte unterteilt und während der Abarbeitung des Vorgangs der Schrittzähler permanent erhöht, sobald ein Einzelschritt abgearbeitet wurde. Der Schrittzähler wird in der globalen Variable recoveryStatus gespeichert. Die Funktion recoverSystem() die bei jedem Starten der Lagersteuerung in main() aufgerufen wird, überprüft durch Einlesen der temporären Log-Datei, ob das Recovery durchgeführt werden muss.

Folgende Tabelle zeigt den Recoverystatus nach dem Abarbeiten der Einzelschritte in den verschiedenen Vorgängen:

	Bestandsanfrage	Verfügbarkeitsanfrage	Anfrage auf freie Stellplätze	Einlagern	Auslagern	Herunterfahren
vorheriger Befehl abgearbeitet	0	0	0	0	0	0
neues Kommando erhalten	1	1	1	1	1	1
A001 gesendet	2	2	2	2	2	2
Kommando an Lager gesendet				3	3	
A001 von Lager erhalten				4	4	
A002 / F001 von Lager erhalten				5	5	
Bestandsdatei aktualisiert				6	6	
A002 / F00x gesendet	0	0	0	0	0	0

Abbildung 11.7: Mögliche Zustände im Recovery

11.2.5.2 Überprüfen des Recoverystatus im Programmcode

Im Programmcode muss bei der Abarbeitung eines Vorgangs für jeden Einzelschritt überprüft werden, ob dieser aufgrund des aktuellen Recovery-Status ausgeführt werden darf. Wird der Einzelschritt ausgeführt, wird der Schrittzähler um 1 erhöht und in die temporäre Log-Datei geschrieben.

Beispiel:

Listing 11.2: Pseudocode für Recovery

```
// Wenn vorheriger Vorgang vollständig abgearbeitet wurde
if (recoveryStatus == 0) {
    recoveryStatus++; // recoveryStatus jetzt 1
    logge recoveryStatus; // schreibt Zähler in temp. Log-Datei
```

11.2 Lagersteuerung

```
}

else {
    überspringeEinzelschritt;
}

// Wenn bei zuvor abgebrochenen Vorgang noch kein A001 gesendet wurde
if (recoveryStatus <= 1) {
    sendeA001;
    recoveryStatus++; // recoveryStatus jetzt 2
logge recoveryStatus; // schreibt Zähler in temp. Log-Datei

}

else {
    überspringeEinzelschritt;
}

...

// Wenn bei zuvor abgebrochenen Vorgang Bestandsdatei aktualisiert
wurde
if (recoveryStatus <= 6) {
    sendeA002;
    recoveryStatus = 0; // 0 bedeutet Einlagervorgang abgeschlossen
leereLogTemp;
}
```

11.2.5.3 Systemstart im Recovery-Modus

Wird beim Starten der Lagersteuerung durch Auslesen der temporären Log-Datei festgestellt, dass der letzte Vorgang nicht vollständig abgearbeitet wurde, muss der entsprechende Befehl der Steuerung nach der letzten gemerkten Schrittmarke fortgesetzt werden.

Folgende Einträge in der temporären Log-Datei wären z.B. möglich:

```
//1
STS1K0051151416354:000004g001
//2
S1STA0011151416354:000000
//3
S1laK0021151416454:000003004
//4
laS1A0011151416454:000000
//5
laS1A0021151416454:000000
//6
```

Der letzte Auslagervorgang wurde nach Schritt 6 (schreiben der Bestandsdatei) abgebrochen. Der Vorgang muss mit recoveryStatus = 6 erneut ausgeführt werden, vorausgesetzt der Benutzer

11.2 Lagersteuerung

wünscht dies. Dazu wird in der Funktion recoverSystem() die Funktion awaitCommand() mit dem gelogten Steuerungskommando aufgerufen. Dadurch wird das Empfangen des entsprechenden Kommandos von der Steuerung simuliert.

Alle Einzelschritte des Auslagervorgangs werden bis einschließlich Schritt 6 bei der Abarbeitung übersprungen. Der noch ausstehende Schritt, ein A002 an die Steuerung zu senden, wird abgearbeitet. Anschließend wird die Variable recoveryStatus auf 0 zurückgesetzt.

11.2.6 Beschreibung von Logging

11.2.6.1 Überblick

Die Klasse Logging schreibt Nachrichten und Statusinformationen in die Logdateien. Diese werden für das Recovery, aber auch zur manuellen Fehlersuche benötigt. Die log_temp.txt enthält den letzten in Abarbeitung befindlichen Auftrag, ist dieser abgearbeitet wird dieser an die log.txt angehängt und die log_temp.txt geleert.

11.2.6.2 Anhängen von log_temp.txt an log.txt

Intern verwendet Logging dafür einen Schrittzähler, der bei jedem Kommando um zwei erhöht wird und bei jedem nicht-Kommando um eins erniedrigt wird. Ist der Schrittzähler gleich null, ist der letzte Auftrag abgeschlossen und die Logdateien werden wie oben beschrieben zusammengeführt.

11.2.6.3 Logging im Recovery Modus

Der Konstruktor von Logging überprüft bei der ersten Instanziierung von Logging, ob die log_temp.txt bereits Einträge enthält, also sich die Lagersteuerung im Recovery Modus befindet. Ist dies der Fall wird der Schrittzähler auf den richtigen Wert gesetzt und anschließend normal fortgefahren.

11.2.6.4 Loggen von Statusinformationen

Für das Loggen der Statusinformationen gibt es eine Helperfunktion in der Helper.cpp, die das Erzeugen eines Objektes, das eigentliche Loggen und den try/catch Block kapselt. Diese Funktion logS(`int status`) gibt bei erfolgreichem schreiben ein true zurück, bei einem Fehler ein false.

11.2.6.5 Loggen von Nachrichten

Das Loggen der Nachrichten geschieht über `writeLog(const string&)`. Diese Methode koordiniert das Schreiben der einzelnen Logdateien und hängt die log_temp.txt zum richtigen Zeitpunkt an die log.txt an.

12 Robotersystem

12.1 Roboter

12.1.1 Einführung

In diesem Abschnitt wird die Roboterzelle kurz vorgestellt sowie alle programmierten Befehle zur Steuerung des Roboters.

Einleitend stellt nachfolgendes Funktionsdiagramm ?? den Programmablauf des Roboters dar:

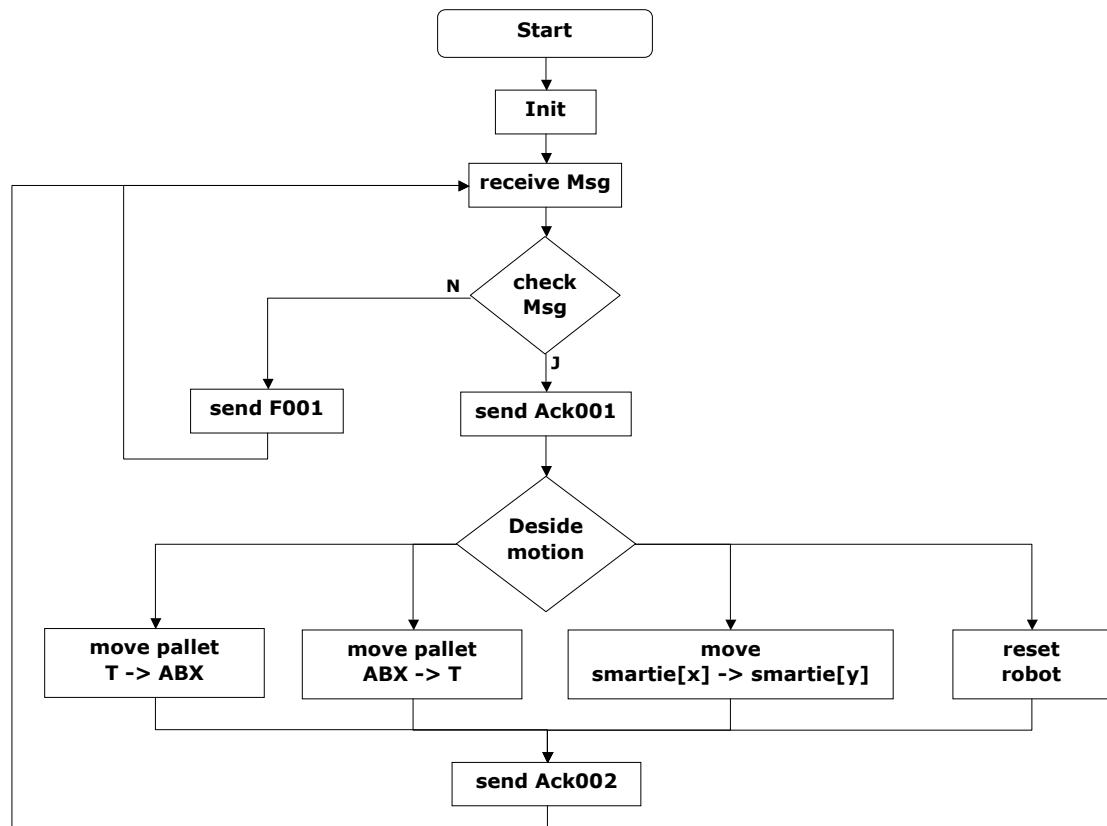


Abbildung 12.1: Funktionsdiagramm Roboterprogramm

Nachfolgendes Bild ?? zeigt die Roboterzelle:

12.1 Roboter



Abbildung 12.2: Roboterarm

12.1 Roboter

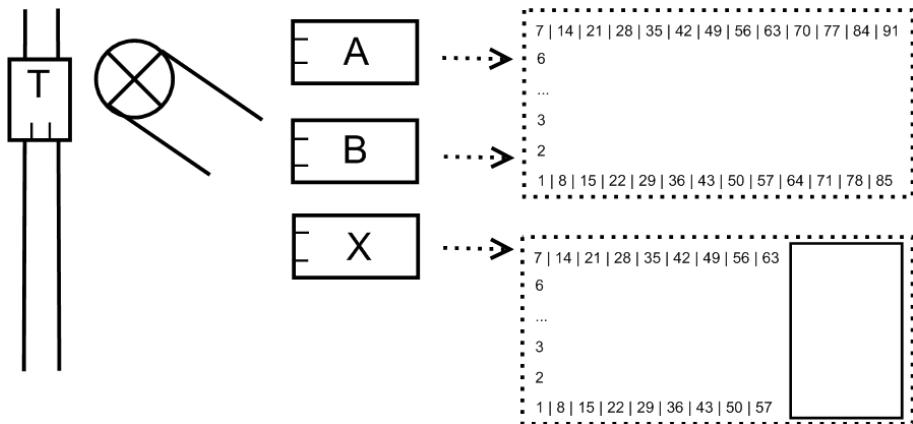


Abbildung 12.3: Zellenlayout

12.1.2 Programmaufbau

Nachfolgend werden die Befehle zur Steuerung des Roboters dargestellt. Das Programm ist in voneinander unabhängige lauffähige Bibliotheken aufgegliedert.

Jede Bibliothek hat eigene globale Variablen sowie Programme (Methoden), welche wiederum lokale Variablen bzw. Parameter besitzen können. Das Hauptprogramm (*fdzRobo*) ruft die entsprechenden Unterprogramme in den Bibliotheken auf. Der Zustandaustausch läuft via Parameterübergabe.

Zu dem Programm (*fdzRobo*) gehören folgende Bibliotheken:

1. *libNetwork*
2. *libCheckCommit*
3. *libDecideMotion*
4. *libMotion*

Jede Bibliothek hat eine eigene Initialisierungssequenz, welche beim Programmstart (*p_CallAllInits*) einmalig aufgerufen wird.

12.1.2.1 Hauptprogramm FDZ Robo

Ergänzung zum Hauptprogramm

Im Hauptprogramm *fdzRobo* beginnt das Bewegungsprogramm der Fabrik der Zukunft. Per Definition vom VAL3 Betriebssystem des Roboters wird standartmäßig nach einem Programm *start* sowie *stop* gesucht. Die Ausführung beginnt im Programm Start und Stop beendet die Ausführung.

12.1 Roboter

Im Start Programm wird ein Task *Main* gestartet, in dem beginnend eine Initialisierung *p_CallAllInits* aufgefufen wird. Die Init stellt in allen libs definierte Anfangszustände her und dient der Userführung am Handbediengerät.

Darauf hin läuft eine Endlosschleife, die dem Programmablauf aus Bild ?? entspricht.

12.1.2.2 libNetwork

In der *libNetwork* wird die gesamte Kommunikation mit der übigerordnetten Robotersteuerung abgewickelt.

Die libNetwork hat folgende Progarme:

1. *pInitNetwork* (Hier wird die Initialisierung der lib durchgeführt.)
2. *pTcpRead* (Methode liest n-Zeichen vom Tcp Socket und übergibt einen String)
3. *pSendAck001* (Methode sendet Elementarbefehl verstanden, benötigt als Übergabeparameter die <Message ID>)
4. *pSendAck002* (Methode sendet Elementarbefehl wurde ausgeführt, benötigt als Übergabeparameter die <Message ID>)
5. *pSendF000* (Methode sendet Elementarbefehl wurde nicht ausgeführt, benötigt als Übergabeparameter die <Message ID>)
6. *pSendF001* (Methode sendet Elementarbefehl wurde nicht ausgeführt, benötigt als Übergabeparameter die <Message ID>)

12.1.2.3 libDecideMotion

In der libDecideMotion befinden sich folgende Programme:

1. *p_MoveABXtoT*
2. *p_MoveTtoABX*
3. *p_MoveSmartie*
4. *p_MoveReset*
5. *p_CalcFrame*
6. *p_CalcPoints*
7. *p_CheckSmartieNr*
8. *p_Decide*
9. *p_InitDecMotion*
10. *p_UserPage*

12.1 Roboter

Das wichtigste Programm ist *p_Decide*, hier wird an Hand des headers entschieden welcher der 4 Fahrbefehle ausgeführt wird

Die 4 Fahrbefehle des Roboters sind unter den Programmern mit Move abgelegt. Im Programmnamen sieht man schon, welche Fahrbewegungen der Roboter ausführen soll. In *p_MoveReset* befindet sich der Befehl, dass der Roboter auf seine Ursprungsposition fahren soll und im Anschluss daran die Armleistung deaktiviert wird.

Im letzten Programm *p_UserPage* werden die ausgeführten Fahrbefehle gespeichert.

In *p_CalcFrame* wird das Frame (s.Abb.)und in *p_CalcPoints* werden die Smartiepositionen berechnet.

p_CalcPoints ist das Initialisierungsprogramm, darin erfolgen die Aufrufe der beiden Kalkulationsprogramme.

Im nächsten Programm *p_CheckSmartieNr* wird geprüft, ob für die jeweilige Palettenart auch die richtige Palettengröße geschickt wurde.

12.1.2.4 libMotion

Enthält folgende Unterprogramme:

1. *p_InitMotion*
2. *p_PickSmartie*
3. *p_PickTray*
4. *p_PlaceSmartiet*
5. *p_PlaceTray*
6. *p_Ursprungpos*

Am Anfang wird den jeweiligen Werkzeugen ein Ventil zugewiesen, dies erfolgt über valve. Des Weiteren macht es Sinn noch die Anfangszustände der Ventile zu definieren

Die nächsten zwei Programme sind dafür da, um einmal Smarties und Trays zu greifen. Die anderen beiden legen die Smarties und die Trays wiederum auf ihre Position, abhängig vom Header ab. Mit Hilfe von appro Befehlen soll eine genaue Anfahrt an das jeweilige Objekt passieren.

Im letzten Programm wurde definiert, wie schnell der Roboter zu seiner Ursprungsposition fahren soll. Die Ausgangssituation ist jHome (0,0,0,0,0,0)

12.1.2.5 libCheckCommit

libCheckCommit enthält nur ein Unterprogramm. Das Programm haben wir *pCheck* genannt. Dort wird der Header in seine Einzelteile zerlegt und nach Richtigkeit geprüft. Hat die Software den Header richtig verstanden, dann schicken wir Acknowledge 001 an die Steuerung. Hingegen besteht ein Übertragungsfehler, dann senden wir die Fehlermeldung F000.

12.1.2.6 libInit

In dieser Bibliothek werden alle Anfangszustände definiert. Einmal welche Ventile was für ein Werkzeug steuern. Des Weiteren wurde bestimmt, dass die Ventile beim Start keine Druckluft durchlassen. Mit Hilfe einer kleinen Wartezeit kann man auch sicher sein, dass die Ventile geschlossen sind. Außerdem werden noch die 4 Frames berechnet und die Ausgangsposition des Roboters geprüft.

12.1.3 Adressierung der Produktmatrix und der Lagermatrizen

Die zu bestückende Produktmatrix besteht aus 63 Feldern, welche von 1 bis 63 durchnummeriert sind. Abbildung ?? zeigt eine solche Matrix.

Weiterhin existieren Lagermatrizen mit 91 Feldern (Siehe Abbildung ??). Auf diesen von 1-91 durchnummerierten Matrizen werden die Smarties gelagert. Der horizontale und vertikale Abstand von der Mitte eines Smarite zur Mitte des benachbarten Smartie beträgt 17,5 mm.

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	49
50	51	52	53	54	55	56
57	58	59	60	61	62	63

Abbildung 12.4: Palette mit 64 Smarties

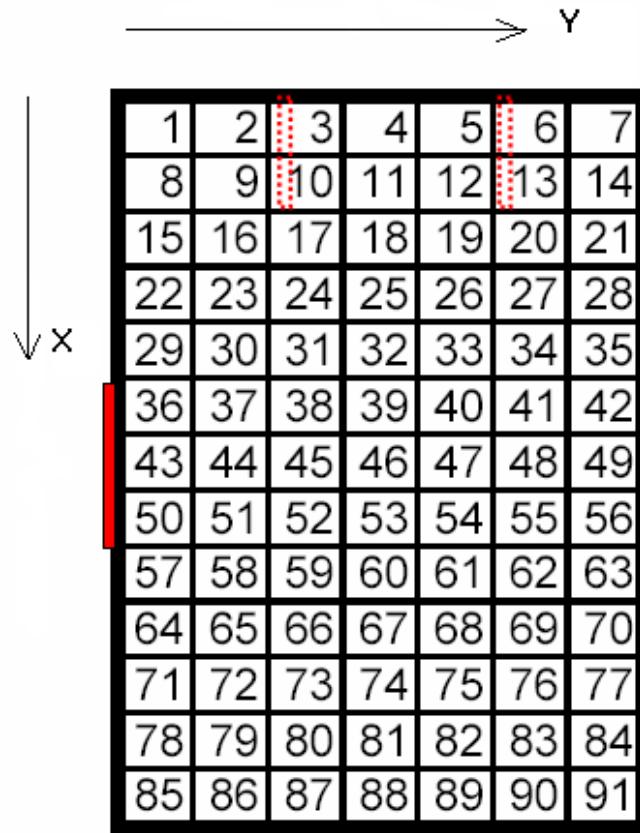


Abbildung 12.5: Palette mit 91 Smarties

12.1.4 Frame-Definition

Es wurden vier Frames angelegt *fPalA* , *fPalB* , *fPalX* und *fPalT* . Jedes Frame definiert sich durch 3 Punkte Abbildung ???. Hier wird das Frame *fPalA* als Beispiel herangezogen. Es wird über die drei Punkte *p0_A* , *pX_A* und *pY_A* definiert. Um z.B. das Frame *fPalA* zu ändern sind folgende Schritte notwendig:

1. Die Punkte *p0_A* , *pX_A* und *pY_A* (Points in libDecideMotion) wie gewohnt teachen und abspeichern.
2. Beim Programmstart die Frage: Wurden Punkte der Frames neu geteached? mit JA bestätigen.
3. Dann wird das Frame berechnet unter Zuhilfenahme der neu geteachten Punkte.

p0_A	pY_A						
	1	2	3	4	5	6	7
	8	9	10	11	12	13	14
	15	16	17	18	19	20	21
	22	23	24	25	26	27	28
	29	30	31	32	33	34	35
	36	37	38	39	40	41	42
	43	44	45	46	47	48	49
	50	51	52	53	54	55	56
pX_A	57	58	59	60	61	62	63
	64	65	66	67	68	69	70
	71	72	73	74	75	76	77
	78	79	80	81	82	83	84
	85	86	87	88	89	90	91

Abbildung 12.6: Frameerstellung

12.1.5 Tool-Definition

Die Greifeinheit am Roboter besitzt zwei Werkzeuge. Diese Werkzeuge werden im folgenden als Tools bezeichnet. Jedes der beiden Tools hat einen eigenen Tool Center Point (sog. TCP).

Auf den nachfolgenden Bildern wird jeweils exemplarisch ein Koordinatensystem an die Stelle des TCP gehalten. Die Achsen des kleinen, farbigen Koordinatensystems sind wie folgt definiert:

- x-Achse -> blau
- y-Achse -> grün
- z-Achse -> rot

Die beiden Tool Center Punkte wurden wie folgt benannt:

1. tGripper

Die Werkzeugorientierung des Greifers ist auf Abbildung ?? zu sehen.

12.1 Roboter

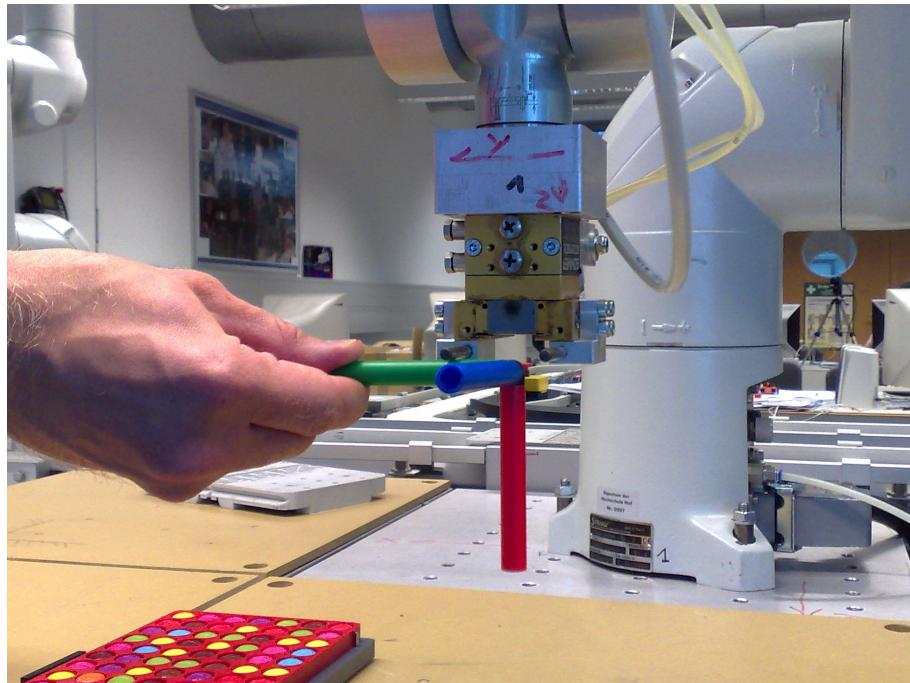


Abbildung 12.7: Tool tGripper

2. tSauger

Die Werkzeugorientierung des Saugers ist auf Abbildung ?? zu sehen.

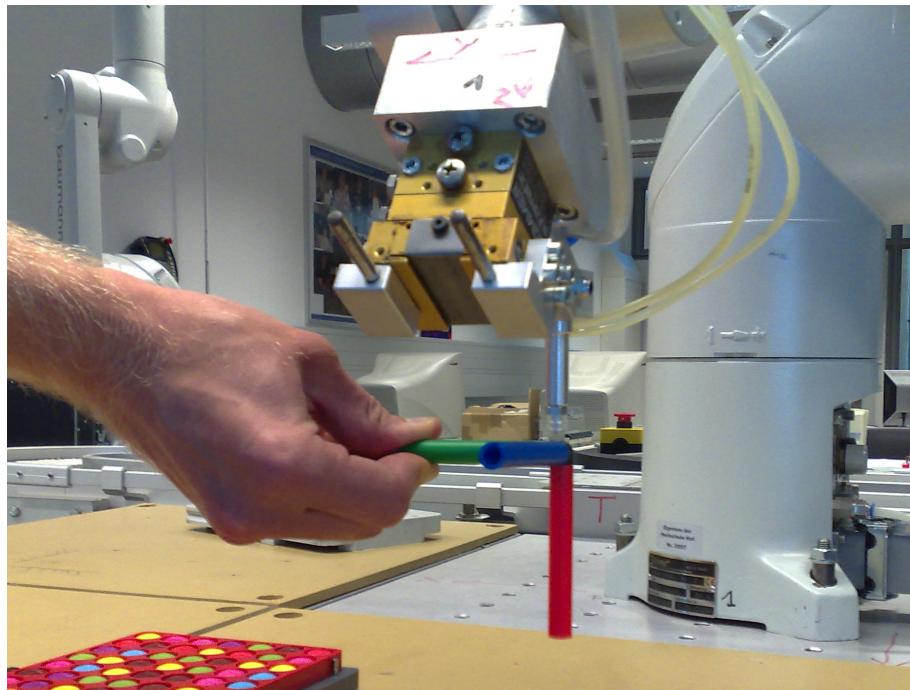


Abbildung 12.8: Tool tSauger

12.2 Implementierung der Robotersteuerung

Mit dem Tool *tGripper* wird die Produktmatrix sowie die Lagermatrix gegriffen und verfahren. Das Tool *tSauger* dient dazu Smarties von einer Palette in die andere zu sortieren.

12.2 Implementierung der Robotersteuerung

12.2.1 main

Vor dem Ansprung von `main` wird je ein Objekt `RoboterSteuerung` und `RoboterSteuerungState` erzeugt, die Logging übernehmen und den Zustand der Steuerung speichern.

`main` registriert als erstes `callback_Trace()`.

Dann werden mit `registerCallbacksControl()` und `registerCallbacksRoboter()` die callbacks registeriert, die die Steuerungslogik abbilden.

Anschliessend wird das Netzwerk mit `RoboterSteuerung::initNetwork()` initialisiert.

Dann wird `wird` mit `doInitialRecovery()` das Recovery behandelt.

Anschliessend wird das System in seine Endlosschleife für den „normalen“ ablauf mit `awaitCommand()` versetzt.

12.2.2 Globale Funktionen

12.2.2.1 fdzRoboterSteuerung.cpp::doInitialRecovery

Prototyp:

- `void doInitialRecovery(void)`

Parameter:

- `void`

Rückgabewert:

- `void`

Bemerkungen:

- Diese Methode erkennt, ob ein Recovery nach dem Start der Robotersteuerung notwendig ist, und führt es aus. Folgende Aktionen werden durchgeführt:

1. Abrufen des letzten empfangenen Kommandos (eine Nachricht mit 'K' als Typ) mit `Recovery::GetLastCommand()` auf dem Steuerungslogfile
2. Prüfen, ob das Kommando mit A001: „Acknowledge 1“ bestätigt wurde.
3. Prüfen, ob das Kommando noch nicht mit A002: „Acknowledge 2“ beantwortet wurde.

12.2 Implementierung der Robotersteuerung

Wenn die Prüfungen erfolgreich waren, wird der Benutzer gefragt, ob das Recovery ausgeführt werden soll.

Wenn ja, wird das Kommando, wie ein „normal“ empfangenes an `parseMessage()` übergeben. Die registrierten Callbacks übernehmen dann die Abarbeitung, wie gehabt. Die eigentliche „Intelligenz“ des Recovery steckt dabei in `doSendToRobot()`. Vergleiche auch Recoveryverhalten Roboter

Wenn der Benutzer den Abbruch wünscht, werden alle Recovery-Files gelöscht und das System geht in `awaitCommand()`.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.2 Network.cpp::connectToControl

Prototyp:

- **void** connectToControl()

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Ruft solange openConnection(), bis eine Verbindung erfolgreich aufgebaut werden konnte.
Der zurückgegebene SocketDescriptor wird in connID_Control gespeichert.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.3 Network.cpp::connectToRobot

Prototyp:

- **void** connectToRobot ()

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Erstellt mit `initServerSocket()` einen ServerSocket und speichert den zurückgegebenen SocketDescriptor in `connID_RobotConnect`. Wartet anschliessend mit `awaitConnection()`, bis eine Verbindung erfolgreich aufgebaut wurde. Der zurückgegebene SocketDescriptor wird in `connID_Robot` gespeichert.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.4 Network.cpp::awaitCommand

Prototyp:

- **void** awaitCommand()

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Wartet in einer Endlosschleife mit `rawRecieveFromControl()` auf Nachrichten, logged diese mit `RoboterSteuerungState::logMessage()` und vermittelt sie anschliessend an `parseMessage()`.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.5 Network.cpp::doSendToControl

Prototyp:

- **void** doSendToControl(SCallbackData* apCallBackData)

Parameter:

- SC callbackData* apCallBackData apCallBackData->message enthält als SMessage die zu sendende Nachricht.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion wickelt den vollständigen Vorgang zum Versenden einer Nachricht an die Steuerung ab.
- Jede übergebene Nachricht wird per rawSendToControl() an die Steuerung gesendet, sofern sie noch nicht im Logfile vorhanden ist.
- Wenn es sich um ein A002: „Acknowledge 2“ handelt, wird das Steuerungslogfile gelöscht.
- **ACHTUNG!!!** Es wird nicht geprüft, ob das A002: „Acknowledge 2“ auch tatsächlich für die Steuerung bestimmt war. Dies ist nur durch die Tatsache festgelegt, dass doSendToControl() als Callback für Nachrichten, die an die Steuerung gehen, registriert ist.
- **ACHTUNG!!!** Diese Funktion geht davon aus, dass lediglich Nachrichten vom Typ A: „Antwort“ oder F: „Fehler“ an die Steuerung gesendet werden. Deswegen wird (anders, als in doSendToRobot()) lediglich geprüft, ob die Nachricht nicht bereits geloggt wurde. Wenn Sie noch nicht geloggt war, wird sie in rawSendToControl() versendet und mit RoboterSteuerungState::logMessage() geloggt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.6 Network.cpp::doSendToRobot

Prototyp:

- **void** doSendToRobot (SCallbackData* apCallBackData)

Parameter:

- SCallbackData* apCallBackData apCallBackData->message enthält als SMessage die zu sendende Nachricht.

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion wickelt den vollständigen Vorgang zum Versenden einer Nachricht an den Roboter ab.
- Als erstes wird geprüft, ob die Nachricht bereits im Logfile ist. Wenn Sie im Logfile ist, wird der Timestamp des apCallBackData->message durch den der gelogten Nachricht ersetzt (Das ist notwendig, damit das Zielsystem, sofern es die Nachricht bereits erhalten hat, diese zuordnen kann). Wenn nicht, wird sie geloggt.
- Dann wird überprüft, ob sich das zu apCallBackData->message gehörige A001: „Acknowledge 1“ im Logfile befindet. Wenn nicht, wird die Nachricht per rawSendToRobot () versendet und auf das A001: „Acknowledge 1“ gewartet.
- Dann wird überprüft, ob das zu apCallBackData->message gehörige A002: „Acknowledge 2“ bereits geloggt ist. Wenn nein, wird auf das zu A002: „Acknowledge 2“ gewartet.
- **ACHTUNG!!!** Es wird nicht geprüft, ob das vom Netz gelesene A001: „Acknowledge 1“ und A002: „Acknowledge 2“ auch tatsächlich zu apCallBackData->message gehört. Das sollte verbessert werden.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.7 Network.cpp::rawSendToControl

Prototyp:

- **void** rawSendToControl (SMessage* apMessageToSend)

Parameter:

- SMessage* apMessageToSend Die zu versendene SMessage.

Rückgabewert:

- **void**

Bemerkungen:

- Versucht solange eine Nachricht per sendMessage() an Steuerung zu versenden, bis das Senden erfolgreich war.
- bei Verbindungsabbrüchen wird solange reconnected, bis die Verbindung wieder steht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.8 Network.cpp::rawSendToRobot

Prototyp:

- **void** rawSendToRobot (SMessage* apMessageToSend)

Parameter:

- SMessage* apMessageToSend Die zu versendene SMessage.

Rückgabewert:

- **void**

Bemerkungen:

- Versucht solange eine Nachricht per sendMessage () an den Roboter zu versenden, bis das Senden erfolgreich war.
- bei Verbindungsabbrüchen wird solange reconnected, bis die Verbindung wieder steht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.9 Network.cpp::rawRecieveFromControl

Prototyp:

- **void** rawRecieveFromControl (SMessage* apMessageToSend)

Parameter:

- SMessage* apMessageToSend Zeiger auf SMessage, in der die empfangene Nachricht abgelegt werden soll.

Rückgabewert:

- **void**

Bemerkungen:

- Versucht solange eine Nachricht per receiveMessage () von Steuerung zu empfangen, bis das Empfangen erfolgreich war.
- bei Verbindungsabbrüchen wird solange reconnected, bis die Verbindung wieder steht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.2.10 Network.cpp::rawRecieveFromRobot

Prototyp:

- **void** rawRecieveFromRobot (SMessage* apMessageToSend)

Parameter:

- SMessage* apMessageToSend Zeiger auf SMessage, in der die empfangene Nachricht abgelegt werden soll.

Rückgabewert:

- **void**

Bemerkungen:

- Versucht solange eine Nachricht per receiveMessage() vom Roboter zu empfangen, bis das Empfangen erfolgreich war.
- bei Verbindungsabbrüchen wird solange reconnected, bis die Verbindung wieder steht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2.3 Die Klasse RoboterSteuerungState

Diese Klasse beinhaltet Funktionen zum Logging und Speichert die Information, welche Palette gerade wo am Roboter steht, und wie diese Palette belegt ist.

ACHTUNG !!! Das Recovery geht davon aus, dass nur der übergebene Stand der Paletten gespeichert wird. Da die Intelligenz für das Recovery bei `doSendToRobot()` liegt, darf eine Veränderung der Paletten während der Bearbeitung NICHT gespeichert werden, sonst scheitert das Recovery!!!

Vergleiche Änderungen entgegen dem allgemeinen Recovery-Verhalten.

12.2.3.1 RoboterSteuerungState::logMessage

Prototyp:

- `void logMessage (SMessage* apSMessage) ;`

Parameter:

- `SMessage* apSMessage`: SMessage die ins Logfile geschrieben werden soll.

Rückgabewert:

- `void`

Bemerkungen:

- Schreibt die Übergebne Nachricht ins betroffene Logfile. Vergleiche Benennungen der Recovery-Files
Wenn die Nachricht an oder vom Roboter war, wird die Nachricht im Roboterlogfile gespeichert.
Wenn die Nachricht an oder vom Roboter war, wird die Nachricht im Steuerungslogfile gespeichert.
In jedem Falle wird die Nachricht im persistenten Logfile gespeichert.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.2 RoboterSteuerungState::setStateOfPlace

Prototyp:

- **void** setStateOfPlace(**char** position, **bool** state)

Parameter:

- **char** position: Position ([ABX]), die geändert werden soll
- **bool** state: **true**: belegt; **false**: frei;

Rückgabewert:

- **void**

Bemerkungen:

- Setzte den Status der betreffenden Position.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.3 RoboterSteuerungState::getStateOfPlace

Prototyp:

- **bool** getStateOfPlace(**char** position) **const**

Parameter:

- **char** position: Position ([ABX]), die abgefragt werden soll

Rückgabewert:

- **bool**: **true**: belegt; **false**: frei;

Bemerkungen:

- Fragt den Status der betreffenden Position ab.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.4 RoboterSteuerungState::isResourcePlaceFree

Prototyp:

- **bool** isResourcePlaceFree() **const**

Parameter:

- **void**

Rückgabewert:

- **bool**: **true**: frei; **false**: belegt;

Bemerkungen:

- Fragt ab, ob KEINE Ressourcenpalette am Roboter steht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.5 RoboterSteuerungState::getPositionOfResourcePalette

Prototyp:

- `char getPositionOfResourcePalette() const`

Parameter:

- `void`

Rückgabewert:

- `char`: [AB] Entweder die Kennzeichnung des Platzes, an dem eine Ressourcenpalette steht, oder '' wenn keine Ressourcenpalette am Roboter steht

Bemerkungen:

- Fragt ab, wo eine Ressourcenpalette am Roboter steht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.6 RoboterSteuerungState::getFreeResourcePalettePosition

Prototyp:

- **char** getFreeResourcePalettePosition() **const**

Parameter:

- **void**

Rückgabewert:

- **char**: [AB] Entweder die Kennzeichnung des Platzes, an dem KEINE Ressourcenpalette steht, oder '' wenn beide Plätze und B mit Ressourcenpalette am Roboter belegt sind.

Bemerkungen:

- Fragt ab, wo KEINE Ressourcenpalette am Roboter steht.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.7 RoboterSteuerungState::getRecourcePalette

Prototyp:

- **char** getRecourcePalette() **const**

Parameter:

- **void**

Rückgabewert:

- **char**: Immer ''

Bemerkungen:

- Diese Funktion ist DEPRECATED. Do Not Use!

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.8 RoboterSteuerungState::getProductPalette

Prototyp:

- `char` getProductPalette() `const`

Parameter:

- `void`

Rückgabewert:

- `char`: Immer 'X'

Bemerkungen:

- Diese Funktion ist DEPRECATED. Do Not Use!
- Im Praxisblock wurde vereinbart, dass die Produktpalettenposition IMMER 'X' ist, obwohl die Spezifikation anders lautet. Prinzipiell ist der Code des Roboters in der Lage, von jeder beliebigen Palette zu jeder beliebigen Palette zu bestücken; allerdings kann er auf der Position 'X' nicht alle Positionen der RessourcenPalette erreichen. (Es können jedoch alle Positionen der Produktpalette erreicht werden)
Durch die vereinbarte Einschränkung wurde sowohl der Code der Robotersteuerung einfacher, wie auch die besondere Behandlung der nicht erreichbaren Positionen entfiel.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.9 RoboterSteuerungState::setRecourcePalette

Prototyp:

- **void** setRecourcePalette(**char** position)

Parameter:

- **char** position wird ignoriert

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion ist DEPRECATED. Do Not Use!
Vergleiche RoboterSteuerungState::getProductPalette()

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.10 RoboterSteuerungState::setProductPalette

Prototyp:

- **void** setProductPalette(**char** position)

Parameter:

- **char** position wird ignoriert

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion ist DEPRECATED. Do Not Use!
Vergleiche RoboterSteuerungState::getProductPalette()

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.3.11 RoboterSteuerungState::ClearLogs

Prototyp:

- **void** ClearLogs (**void**)

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion löscht sowohl das Roboterlogfile, als auch das Steuerungslogfile
Vergleiche Benennungen der Recovery-Files

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.4 Die Klasse RoboterSteuerung

Diese Klasse beinhaltet ausgelagerte Funktionen, welche teilweise von den Callback-Routinen aufgerufen werden.

12.2.4.1 RoboterSteuerung::initNetwork

Prototyp:

- **void** initNetwork (**void**)

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion registriert:
 - doSendToControl () als Callback für Nachrichten, die an die Steuerung gesendet werden sollen
 - doSendToRobot () als Callback für Nachrichten, die an den Roboter gesendet werden sollen

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.4.2 RoboterSteuerung::sendMessage

Prototyp:

- **void** sendMessage (SMessage* apSMessage)

Parameter:

- SMessage* apSMessage

Rückgabewert:

- **void**

Bemerkungen:

- Diese Funktion versendet Nachrichten, indem die Nachrichten an `parseMessage()` übergeben werden. Durch entsprechend in `RoboterSteuerung::initNetwork()` registrierten Callbacks werden die Nachrichten an das System gesendet, für das sie gedacht sind.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.4.3 RoboterSteuerung::*splitMessage*

Prototyp:

- `int splitMessage(char *assambleMessage)`

Parameter:

- `char *assambleMessage`: payload-Part der SMessage, der eine Bestückung veranlasst (STSrK002: „Bestücke Produktpalette“)

Rückgabewert:

- `int`

Bemerkungen:

- Diese Methode wird vom Callback für das Kommando STSrK002: „Bestücke Produktpalette.“ verwendet.
Die Methode zerlegt den Bestückungsbefehl in einzelne Elementarbefehle und versendet diese.
Als Überabeparameter wird der Payload des Bestückungsbefehls mitgegeben.
Dazu werden ausserdem folgende Methoden benötigt:

Fehlerbehandlung:

-

Beispiele:

-

12.2 Implementierung der Robotersteuerung

12.2.4.4 RoboterSteuerung::string *toString(int i)*

Diese Methode liefert einen zweistelligen String, der den Übergabeparameter repräsentiert.

12.2.4.5 RoboterSteuerung::int *nextResourcePosition(char *aMessage)*

Hier wird der Payload der Lagerpaletteninformation übergeben, der nächste belegte Platz gesucht und dieser in der Information gelöscht.

12.2.4.6 RoboterSteuerung::int *countColor(char* command)*

Diese Methode gibt die Anzahl der Smatrices des übergebenen Payloads an.

12.2.4.7 RoboterSteuerung::char *getColor(char* command)*

Diese Methode gibt die Farbe der Smatrices des übergebenen Payloads an.

12.2.4.8 RoboterSteuerung::setResourcePalettePayload

Die Methode **void** RoboterSteuerung::setResourcePalettePayload(**char** * storageMessage) wird benötigt, damit die Robotersteuerung die Bestückungsinfo merken kann.

Keins Keins

12.2.5 Die Klasse Recovery

Diese Klasse beinhaltet ausgelagerte Funktionen, welche teilweise von den Callback-Routinen aufgerufen werden.

12.2.5.1 Recovery::Recovery

Prototyp:

- Recovery(**const char*** file_name)

Parameter:

- **const char*** file_name: Name (ggf. und Pfad) des Files, in dem die Klasse ihre Informationen ablegt. Vergleiche Benennungen der Recovery-Files

Rückgabewert:

- Recovery: Eine Instanz der Klasse

12.2 Implementierung der Robotersteuerung

Bemerkungen:

- Konstruktor der Klasse. Wenn das angegebene File existiert, wird der Inhalt ausgelesen.
Wenn es nicht existiert, wird das File angelegt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.5.2 Recovery::PutMessage

Prototyp:

- **bool** PutMessage (**const** SMessage& m)

Parameter:

- **const** SMessage& m: Die Nachricht, die ans Logfile angehängt werden soll

Rückgabewert:

- **bool true**, wenn Erfolgreich, **false** sonst.

Bemerkungen:

- Hängt eine Nachricht ans Logfile an

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.5.3 Recovery::GetLastCommand

Prototyp:

- **bool** GetLastCommand (SMessage* m) **const**

Parameter:

- SMessage* m: Pointer auf die SMessage-Struktur, in die das letzte Kommando (Kxx) hinterlegt werden soll.

Rückgabewert:

- **bool true**, wenn Erfolgreich, **false** sonst.

Bemerkungen:

- Hängt eine Nachricht ans Logfile an.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.5.4 Recovery::IsCommandLogged

Prototyp:

- `bool IsCommandLogged(const SMessage& m) const ;`

Parameter:

- `const SMessage& m`: Die Nachricht, die im Logfile gesucht werden soll

Rückgabewert:

- `bool true`, wenn `m` im Logfile existiert, `false` sonst.

Bemerkungen:

- Sucht eine Nachricht im Logfile.
- Das Vergleichen erfolgt vermittels `compareMessage()`, deswegen werden Wildcards unterstützt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.5.5 Recovery::replaceWithAccordingLoggedCommand

Prototyp:

- `bool replaceWithAccordingLoggedCommand(SMessage& m) ;`

Parameter:

- `SMessage& m`: Die Nachricht, die im Logfile gesucht werden soll

Rückgabewert:

- `bool true`, wenn erfolgreich, `false` sonst.

Bemerkungen:

- Sucht das übergeben Kommando in der Datei und ersetzt den Parameter mit der gefundenen Nachricht
- Das Vergleichen erfolgt vermittels `compareMessage()`, deswegen werden Wildcards unterstützt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2.5.6 Recovery::Kill

Prototyp:

- `bool Kill();`

Parameter:

- `void`

Rückgabewert:

- `bool true`, wenn erfolgreich, `false` sonst.

Bemerkungen:

- Löscht das aktuelle Logfile
- **Achtung:** Nach einem Aufruf von Kill ist der interne Filhandle freigegeben, und das File muss erneut geöffnet werden!
- vergleiche `Recovery::Clear()`

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2.5.7 Recovery::Clear

Prototyp:

- `bool Clear();`

Parameter:

- `void`

Rückgabewert:

- `bool true`, wenn erfolgreich, `false` sonst.

Bemerkungen:

- Löscht das aktuelle Logfile und öffnet es sofort wieder.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2.6 Die Callbacks für Nachrichten von der Steuerung

Generell handeln diese Callbacks das versenden von Acknowledges (A001 und A002) selbst. Dies ist notwendig, da in Antworten der Robotersteuerung (Sr) an die Steuerung (ST) gefordert ist, dass das Kommando ausführbar ist, wenn ein A001 gesendet wurde, und nur der jeweilige Callback die Information zur Ausführbarkeit hat. Ohne diese Forderung könnte prinzipiell das Versenden von A001 und A002 auch einheitlich von `awaitCommand()` gelöst werden.

`awaitCommand()` empfängt die Nachricht STSrK000: „Nehme Lagerpalette vom Transportband.“ von der Steuerung, und dispatched diese an `parseMessage()`.

12.2.6.1 CallbackFunktionsControl.cpp::callback_TakeStoragePaletteFromTransport

Prototyp:

- `void callback_TakeStoragePaletteFromTransport (SCallbackData* apCallbackData)`

Parameter:

- `SCallbackData*` `data`: `SCallbackData` der von `parseMessage()` befüllt wird.

Rückgabewert:

- `void`

Bemerkungen:

- Dies ist der Callback für das Kommando STSrK000: „Nehme Lagerpalette vom Transportband.“
Die Vollständige Abarbeitung erfolgt hier.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.2 CallbackFunktionsControl.cpp::callback_TakeProductPaletteFromTransport

Prototyp:

- **void** callback_TakeProductPaletteFromTransport (SCallbackData* apCallbackData)

Parameter:

- SCallbackData* data: SCallbackData der von parseMessage () befüllt wird.

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando STSrK001: „Nehme Produktpalette vom Transportband.“
Die Vollständige Abarbeitung erfolgt hier.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.3 CallbackFunktionsControl.cpp::callback_AssembleProductPalette

Prototyp:

- **void** callback_AssembleProductPalette(SCallbackData* apCallbackData)

Parameter:

- SCallbackData* data: SCallbackData der von parseMessage() befüllt wird.

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando STSrK002: „Bestücke Produktpalette.“ Die Vollständige Abarbeitung erfolgt unter Zuhilfenahme von RoboterSteuerung::splitMessage() hier.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.4 CallbackFunktionsControl.cpp::callback_PutStoragePaletteToTransport

Prototyp:

- **void** callback_PutStoragePaletteToTransport (SCallbackData* apCallbackData)

Parameter:

- SCallbackData* data: SCallbackData der von parseMessage () befüllt wird.

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando STSrK003: „Stelle Lagerpalette auf das Transportband.“
Die Vollständige Abarbeitung erfolgt hier.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.5 CallbackFunktionsControl.cpp::callback_PutProductPaletteToTransport

Prototyp:

- **void** callback_PutProductPaletteToTransport (SCallbackData* apCallbackData)

Parameter:

- SCallbackData* data: SCallbackData der von parseMessage () befüllt wird.

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando STSrK004: „Stelle Produktpalette auf das Transportband.“
Die Vollständige Abarbeitung erfolgt hier.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.6 CallbackFunktionsControl.cpp::callback_SwitchStoragePaletts

Prototyp:

- **void** callback_SwitchStoragePaletts (SCallbackData* apCallbackData)

Parameter:

- SCallbackData* data: SCallbackData **der von** parseMessage () **befüllt wird.**

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando STSrK005: „Tausche Lagerpaletten.“
Die Vollständige Abarbeitung erfolgt hier.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.7 CallbackFunktionsControl.cpp::callback_ResetRobot

Prototyp:

- **void** callback_ResetRobot (SCallbackData* apCallbackData)

Parameter:

- SCallbackData* data: SCallbackData der von parseMessage () befüllt wird.

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando STSrK006: „Reset/Neustart (Herunterfahren) der Robotersteuerung.“
Die Vollständige Abarbeitung erfolgt hier.
- **ACHTUNG!!!** Das Kommando STSrK006: „Reset/Neustart (Herunterfahren) der Robotersteuerung.“ wird nur mit A001 Quittiert. Deswegen wird in dieser Funktion NICHT, wie in allen anderen in CallbackFunktionsControl.cpp RoboterSteuerung::sendMessage () verwendet, sondern das A001 wird in dieser Funktion abgewartet und das Recoveryhandling wird ebenfalls hier lokal erledigt.

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.8 CallbackFunktionsControl.cpp::callback_Trace

Prototyp:

- **void** callback_Trace (SCallbackData* apCallbackData)

Parameter:

- SCallbackData* data: SCallbackData der von parseMessage () befüllt wird.

Rückgabewert:

- **void**

Bemerkungen:

- Dieser Callback ist mit der Referenznachricht „*“ registriert (d.h. er wird für ALLE Nachrichten ausgeführt, sowohl eingehende, als auch ausgehende). Dadurch wird ein „Tracing“ aller Nachrichten, die durch das System gehen möglich. (Mit Ausnahme von STSrK006: „Reset/Neustart (Herunterfahren) der Robotersteuerung.“ und dem A001 darauf, da in callback_ResetRobot () (siehe dort) NICHT parseMessage () verwendet werden konnte.)

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.6.9 CallbackFunktionsControl.cpp::registerCallbacksControl

Prototyp:

- **void** registerCallbacksControl()

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Registriert alle in CallbackFunktionsControl.cpp hinterlegten Callbacks

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.7 Die Callbacks fuer Nachrichten vom Roboter

12.2.7.1 CallbackFunktionsRoboter.cpp::callback_A001

Prototyp:

- **void** callback_A001 (SCallbackData* aCallBackData)

Parameter:

- SCallbackData* data: SCallbackData der von parseMessage () befüllt wird.

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando roSrA001: „Elementarbefehl wurde verstanden.“
In diesem Callback wird lediglich die Nachricht mit RoboterSteuerungState::logMessage () geloggt.
Zur Begründung vergleiche Die Callbacks fuer Nachrichten von der Steuerung

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.7.2 CallbackFunktionsRoboter.cpp::callback_A002

Prototyp:

- **void** callback_A002 (SCallbackData* aCallBackData)

Parameter:

- SCallbackData* data: SCallbackData **der von** parseMessage () **befüllt wird.**

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando roSrA002: „Elementarbefehl wurde ausgeführt :“
In diesem Callback wird lediglich die Nachricht mit RoboterSteuerungState::logMessage () geloggt.

Zur Begründung vergleiche Die Callbacks fuer Nachrichten von der Steuerung

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.7.3 CallbackFunktionsRoboter.cpp::callback_F000

Prototyp:

- **void** callback_F000 (SCallbackData* aCallBackData)

Parameter:

- SCallbackData* data: SCallbackData **der von** parseMessage () **befüllt wird.**

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando roSrF000: „Elementarbefehl wurde nicht verstanden.“

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.7.4 CallbackFunktionsRoboter.cpp::callback_F001

Prototyp:

- **void** callback_F000 (SCallbackData* aCallBackData)

Parameter:

- SCallbackData* data: SCallbackData **der von** parseMessage () **befüllt wird.**

Rückgabewert:

- **void**

Bemerkungen:

- Dies ist der Callback für das Kommando roSrF000: „Elementarbefehl wurde nicht verstanden.“

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

12.2 Implementierung der Robotersteuerung

12.2.7.5 CallbackFunktionsRoboter.cpp::registerCallbacksRoboter

Prototyp:

- **void** registerCallbacksRoboter()

Parameter:

- **void**

Rückgabewert:

- **void**

Bemerkungen:

- Registriert alle in CallbackFunktionsRoboter.cpp hinterlegten Callbacks

Fehlerbehandlung:

- Keins

Beispiele:

- Keins

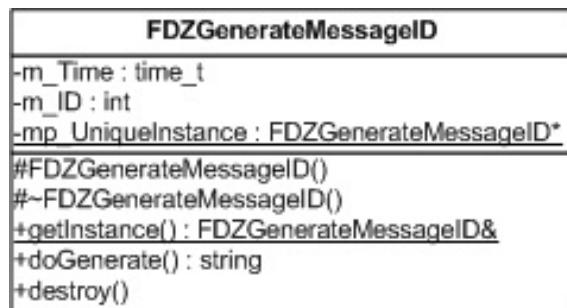
13 Steuerung

13.1 Gemeinsame Klassen

13.1.1 Nachrichtengenerator

13.1.1.1 Klasse FDZGenerateMessageID

Um eine Eindeutigkeit der MessageID innerhalb des FDZNetworks gewährleisten zu können wurde die Klasse **FDZGenerateMessageID** implementiert. Diese erstellt unter Zuhilfenahme des Singleton Patterns die einzigartige MessageID. Die Klasse ist wie folgt aufgebaut:

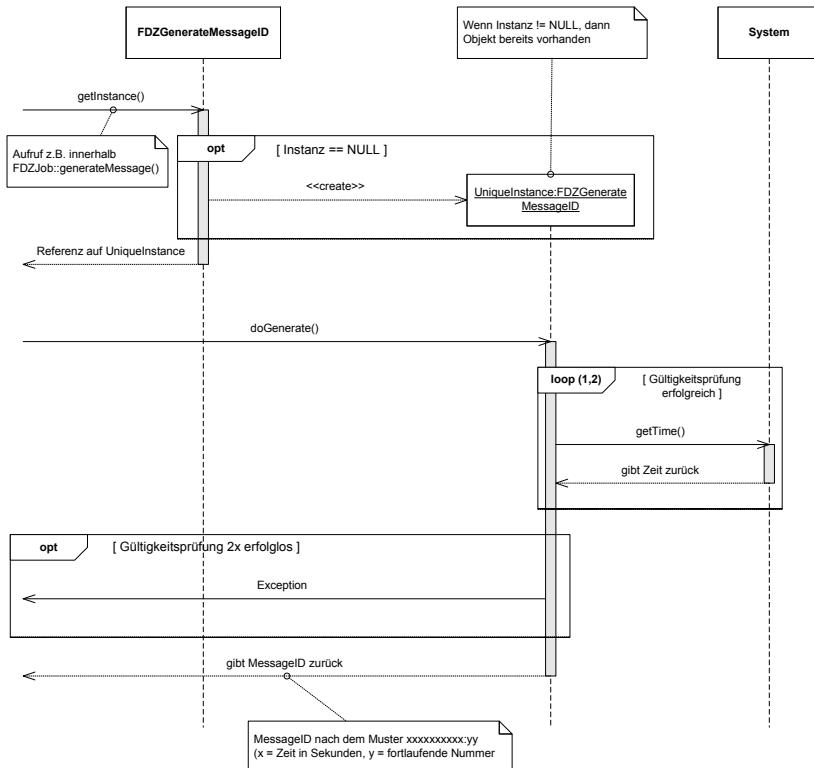


Um mit der Klasse arbeiten zu können muss zunächst die Instanz davon abgerufen werden. Durch den Aufruf der Methode *doGenerate()* erhält man als Rückgabe einen String mit der MessageID.

```
#include "fdz_generate_message_id.cpp"

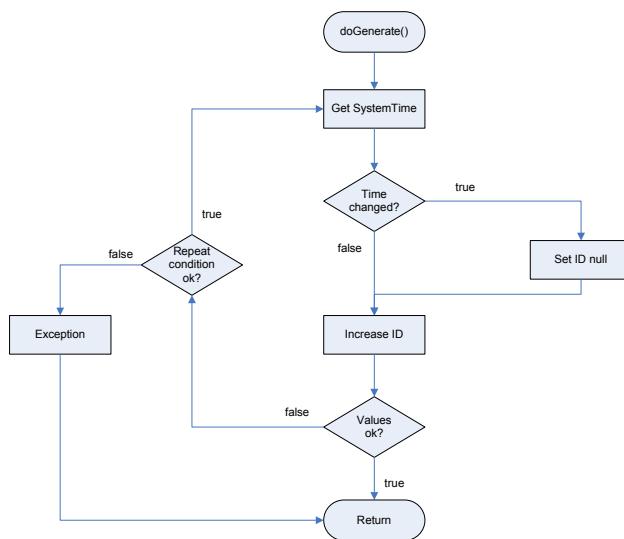
FDZGenerateMessageID& v_Gen = FDZGenerateMessageID::getInstance();
try
{
    string v_Message = v_Gen.doGenerate();
}
catch (FDZException &ex)
{
    //sonstige Fehlerbehandlung
    throw; //Weiterleitung nach oben
}
```

13.1 Gemeinsame Klassen



Zuerst wird die aktuelle Zeit abgerufen und mit der in der Instanz gespeicherten Zeit verglichen. Unterscheidet sich diese, so wird die interne automatisch fortlaufende Nummer zurückgesetzt und die neue Zeit gespeichert. Sollte der Versuch die aktuelle Systemzeit abzurufen fehlschlagen oder die fortlaufende Nummer größer als 99 sein so wird eine Sekunde gewartet und von vorne begonnen. Scheitert auch dieser Versuch erneut wird eine Exception an den Aufrufer der Methode `doGenerate()` geworfen. Ansonsten wird die MessageID als string zurückgegeben.

13.1 Gemeinsame Klassen



Der Speicher der Instanz kann mittels der Methode `destroy()` freigegeben werden. Allerdings darf dies ausschließlich direkt vor dem Herunterfahren von Control geschehen, da ansonsten die Eindeutigkeit der MessageID nicht mehr gewährleistet ist.

13.1.1.2 FDZJob::generateMessage()

Die Methode `generateMessage()` realisiert die Funktionalität der einheitlichen Nachrichtengenerierung für das Modul Control. Diese befindet sich in der abstrakten Klasse `FDZJob`, von der alle einzelnen Submodulklassen abgeleitet werden.

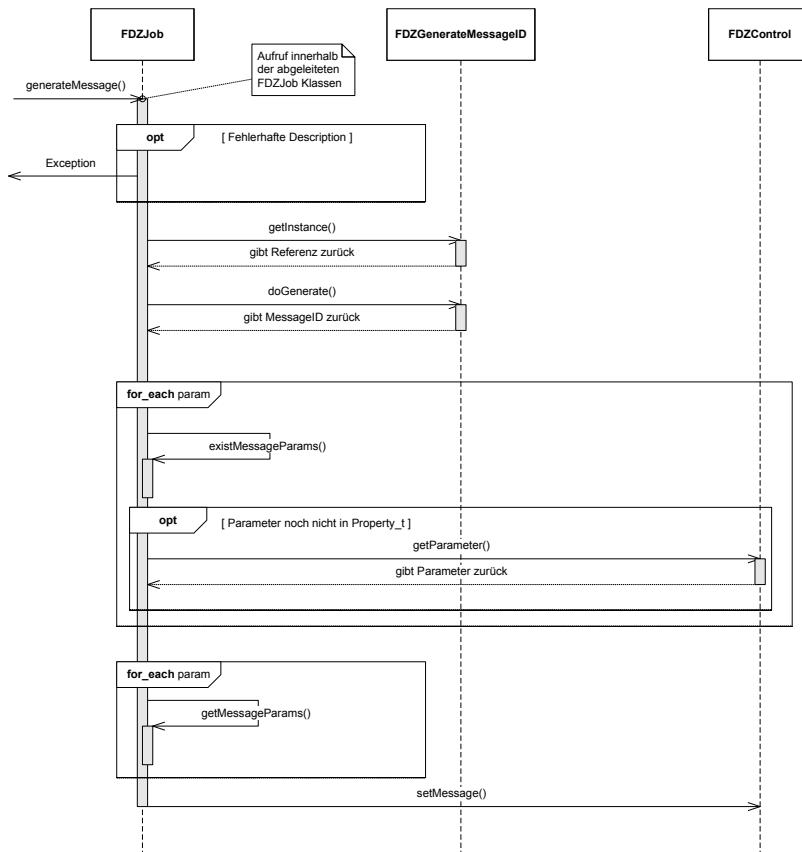
Als Grundlage wird die `description` innerhalb der `FDZJob`-Klasse genommen. Dabei handelt es sich um Beschreibung des Jobs, die bei der Erstellung der jeweiligen Instanz der abgeleiteten Klasse an die JobFactory übergeben wird. Die `description` ist genau 8 Zeichen lang und hat

13.1 Gemeinsame Klassen

immer folgenden Aufbau:

Quelle	2 Zeichen	ST
Ziel	2 Zeichen	Sr; Se; St; Sl
Kommando	1 Zeichen	K
K-Index	3 Zeichen	000; ... ; 006

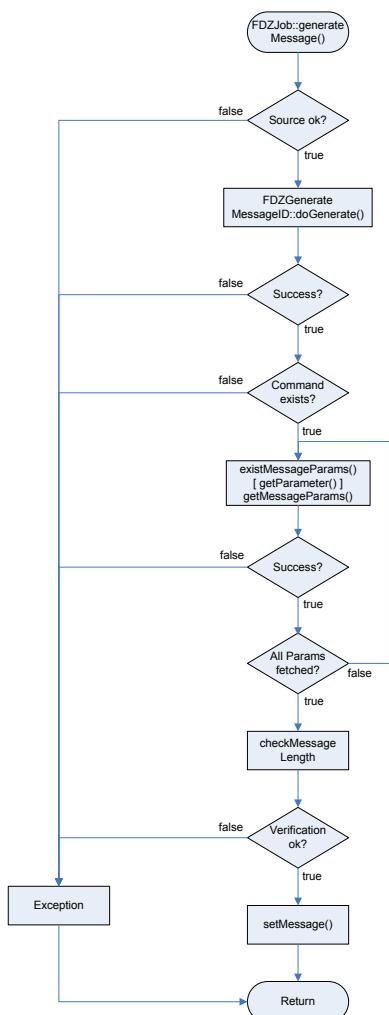
Die Methode hat folgenden Aufbau:



Am Anfang werden Überprüfungen durchgeführt ob die Quelle (Absender), also Control, korrekt ist und ob es sich um ein Kommando handelt. Dann delegiert die Methode Arbeit an die

13.1 Gemeinsame Klassen

Instanz von FDZGenerateMessageID. Nun wird - je nach Befehl - die Nutzdatenlänge für die Nachricht bestimmt und Parameter für Parameter mittels der Methode `existMessageParams()` überprüft, ob das entsprechende Pair bereits in der lokalen Map `m_params` existiert. Ist das nicht der Fall, so wird dieses mit den Daten von FDZControl mit Hilfe von get-Methoden erzeugt und in die lokale Map eingefügt. Dann werden die Nutzdaten - ebenfalls Parameter für Parameter - via `getMessageParams()` abgerufen. Anschließend wird die Gesamtnachricht zusammengesetzt und nochmals auf eine gültige Länge hin überprüft. Tritt während der Abarbeitung ein Fehler auf, z.B. ein ungültiger Befehl oder ein ungültiges Subsystem, so wird eine Exception vom Typ FDZException ausgelöst. Unter Umständen kann auch FDZGenerateMessageID eine Exception werfen, die einfach nach oben weitergereicht wird.



War die Abarbeitung erfolgreich, so wird die Nachricht mit `setMessage()` in FDZControl gespeichert. Die Nachricht, die jetzt über FDZNetwork verschickt werden kann, hat folgenden Aufbau:

Description	8 Zeichen
MessageID	13 Zeichen
Nutzdatenlänge	4 Zeichen
Nutzdaten	<Nutzdatenlänge> Zeichen

13.1.2 FileIO

Die *FDZFileIO*-Klasse übernimmt die Aufgaben der Datei-E/A Funktionen. Alle benötigten Dateizugriffe werden in dieser Klasse gekapselt und anderen Klassen zur Verfügung gestellt. Einerseits kann ein Objekt dieser Klasse mit einer bestimmten Datei verknüpft werden, indem dem Konstruktor ein Dateiname übergeben wird. Andererseits stellt die Klasse auch einfache Funktionen als statische Version zur Verfügung, wie zum Beispiel das Prüfen, ob eine Datei existiert.

Die Klasse *FDZFileIO* integriert eine Überprüfung, ob eine zuvor geschriebene Datei komplett ist. Dies wird damit realisiert, indem bei jedem Schreibzugriff eine symbolische Zeichenfolge an die Datei angehängt wird, welche das Ende der Datei repräsentiert. Somit wird sichergestellt, dass die jeweilige Datei bis zum Ende geschrieben werden konnte. Änderungen, die von außen am Dateiinneren vorgenommen wurden, können allerdings nicht erkannt werden.

Für Objekte der Klasse *FDZFileIO* wird die Variable *Filename* benötigt, die dem Konstruktor übergeben wird. Somit wird das Objekt an die jeweilige Datei gebunden und es kann mit ihr gearbeitet werden. Die Membervariable *EOF* beinhaltet die Zeichenfolge, die für die Überprüfung des Dateiendes verwendet wird.

Die Funktion *fileExists()* überprüft, ob die Datei des aktuellen Objekts existiert und gibt True zurück, wenn dies der Fall ist. Andernfalls wird False zurückgegeben. Die statische Variante *fileExists(string)* überprüft die Existenz einer beliebigen Datei und gibt wie ihr Pendant True oder False zurück. Die nicht-statische Version greift auf diese Variante zurück.

13.1 Gemeinsame Klassen

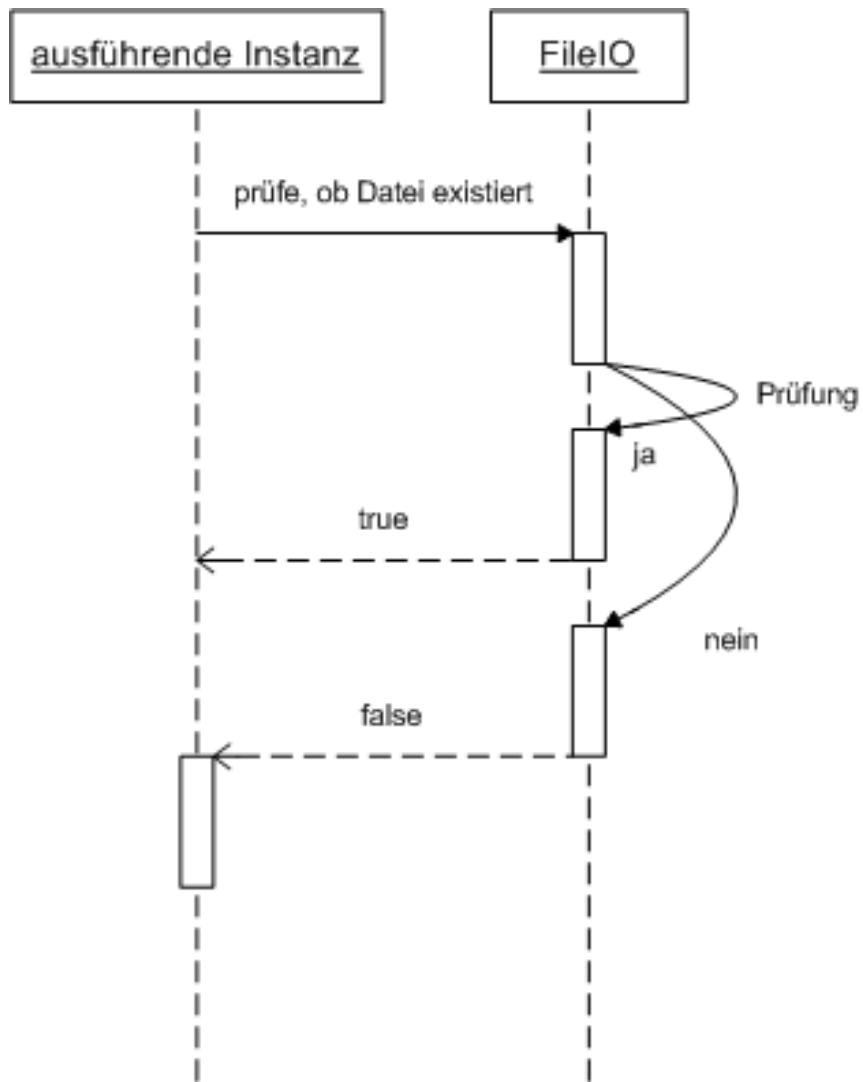


Abbildung 13.1: Abfrage, ob Datei existiert

Eine Datei kann mit der Funktion `removeFile()` gelöscht werden. Auch hier gibt ebenfalls wieder zwei Versionen. Die erste Version, die durch den Konstruktor an eine bestimmte Datei gebunden wurde, `removeFile()` und eine statische `removeFile(string)`, mit der eine beliebige Datei gelöscht werden kann. Auch hier wird wieder die statische Version durch die nicht-statische aufgerufen.

13.1 Gemeinsame Klassen

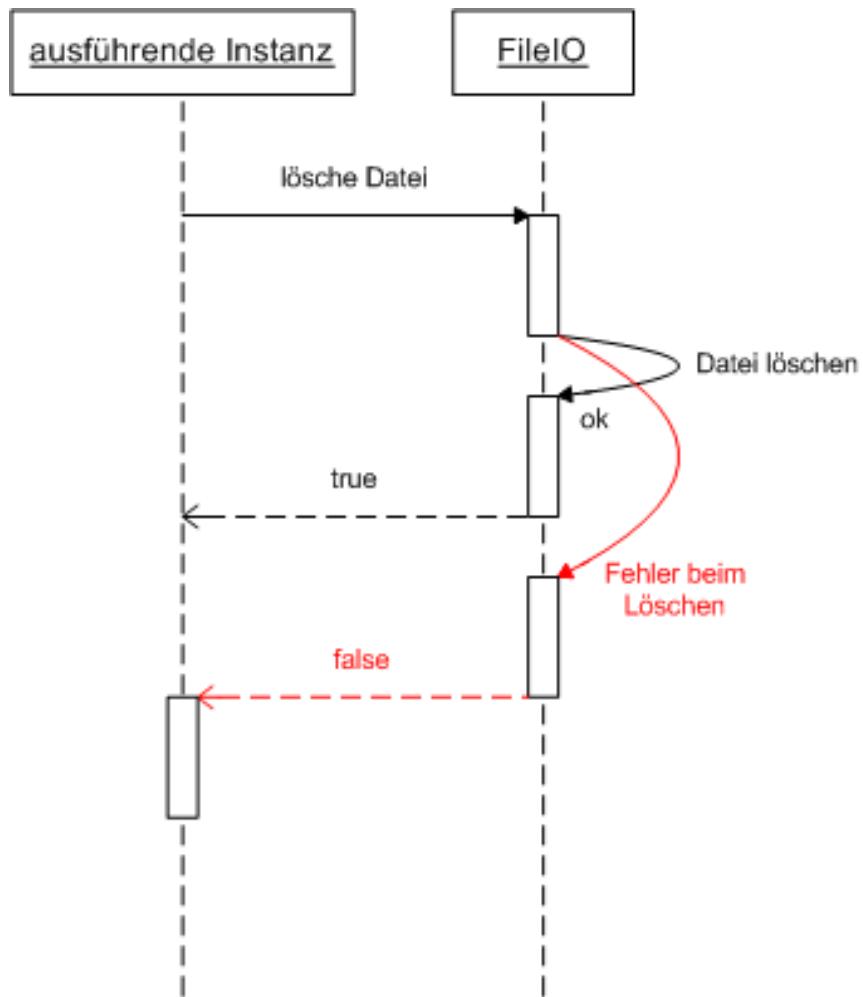


Abbildung 13.2: Datei löschen

Da Control auch Dateien umbenennen muss, bietet die Klasse *FDZFileIO* zwei Methoden zum Umbenennen von Dateien an und heißen *renameFile()* und *renameFile(string)*. Diese arbeiten genauso wie *fileExists()* und *removeFile()*.

13.1 Gemeinsame Klassen

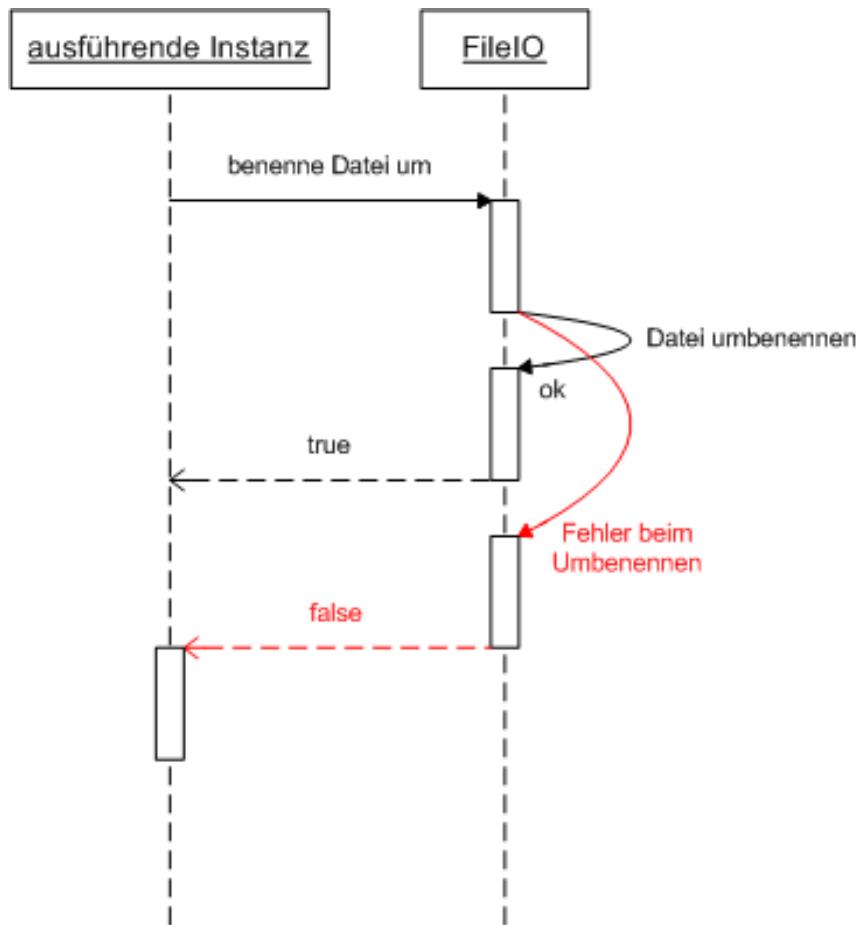


Abbildung 13.3: Datei umbenennen

Zum Schreiben einer Datei wird `writeTo(vector<string>)` verwendet. Bei dieser Funktion wird die komplette Datei neu geschrieben. Dazu wird die Hilfsfunktion `writeFile(vector<string>)` aufgerufen, die sich lediglich um das Schreiben der Datei kümmert. Jeder Eintrag in dem übergebenen Vector wird in eine eigene Zeile geschrieben und zum Schluss das EOF-Flag angehängt. Wenn es beim Schreiben zu einem Fehler kommt, wird der Funktion `writeTo(vector<string>)` False zurückgegeben, sonst wird True zurückgegeben. Sollte die Datei bereits existiert haben, wird diese vor dem Schreiben umbenannt, indem ihr ein „.bak“ angehängt wird. Erst wenn die neue Datei komplett geschrieben wurde, wird die Backup-Datei gelöscht. Sollte es in der Funktion zu einem Fehler kommen, wird eine *FDZException* mit einer entsprechenden Fehlermeldung ausgelöst. Wenn ansonsten alles funktioniert hat, wird True zurückgegeben.

13.1 Gemeinsame Klassen

Allgemein für alle Instanzen, die Dateien brauchen

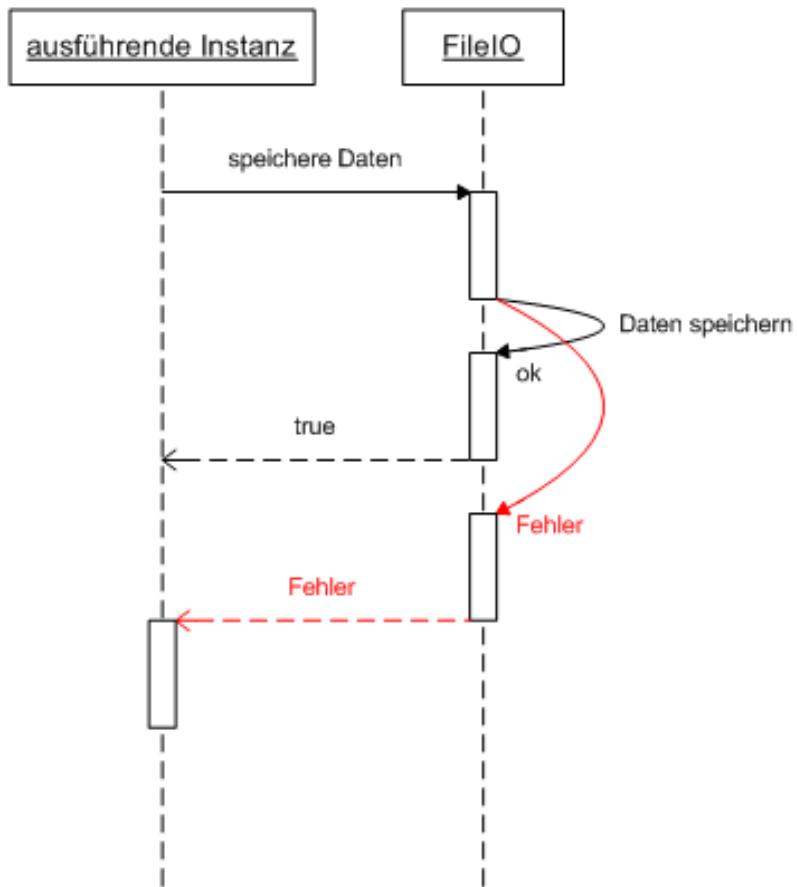


Abbildung 13.4: Datei schreiben

Soll nur eine neue Zeichenfolge zu einer Datei hinzugefügt werden, sollte die Funktion *appendTo(string)* aufgerufen werden. Die Datei wird dann mit der übergebenen Zeichenfolge erweitert und mit dem EOF-Flag abgeschlossen. Sobald ein Fehler auftritt, wird eine *FDZException* mit einer entsprechenden Beschreibung ausgelöst. Ansonsten wird True zurückgegeben.

Für das Einlesen einer Datei ist die Funktion *readFrom(bool)* zuständig. Der Aufrufparameter gibt an, ob versucht werden soll die Datei anhand des Backups wiederherzustellen, falls sie beschädigt sein sollte (Datei nicht vorhanden oder EOF-Flag fehlt). Dieser Parameter ist standardmäßig auf False gesetzt. Wenn das Auslesen geklappt hat, wird True zurückgegeben, ansonsten wird eine *FDZException* mit einer entsprechenden Beschreibung ausgelöst.

13.1 Gemeinsame Klassen

Allgemein für alle Instanzen, die Dateien brauchen

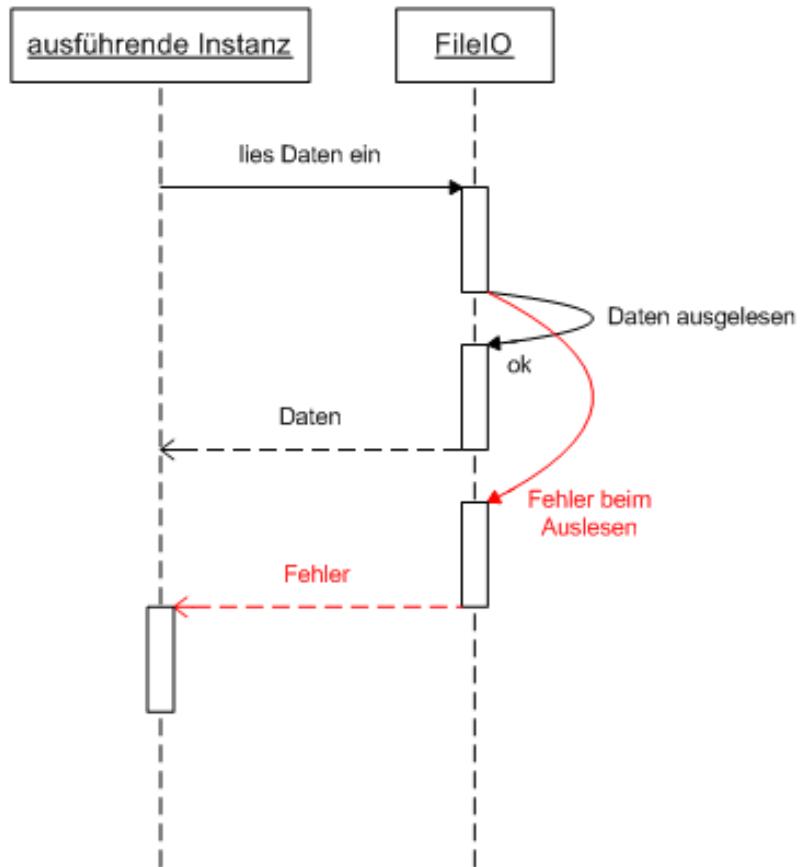


Abbildung 13.5: Datei lesen

Ist die Datei beschädigt und es soll versucht werden die Datei wiederherzustellen, wird überprüft, ob es die Datei mit dem Namen *Filename* mit der Erweiterung „.bak“ gibt. Ist dies nicht der Fall, ist ein Backup nicht möglich. Ansonsten wird zuerst geprüft, ob die Backup-Datei gültig ist. Anschließend wird der letzte Eintrag aus der fehlerhaften Datei genommen und es wird geprüft, ob dieser auch in der Backup-Datei vorhanden ist. Wird eine Übereinstimmung gefunden, wird der Rest der Backup-Datei genommen. Ein Backup ist jedoch nur dann möglich, wenn nur am Anfang einer Datei Änderungen vorgenommen wurden und Teile des unveränderten Inhalts geschrieben werden konnten, so dass es in den beiden Dateien zu Überschneidungen kommt, die dann verglichen werden können.

13.1 Gemeinsame Klassen

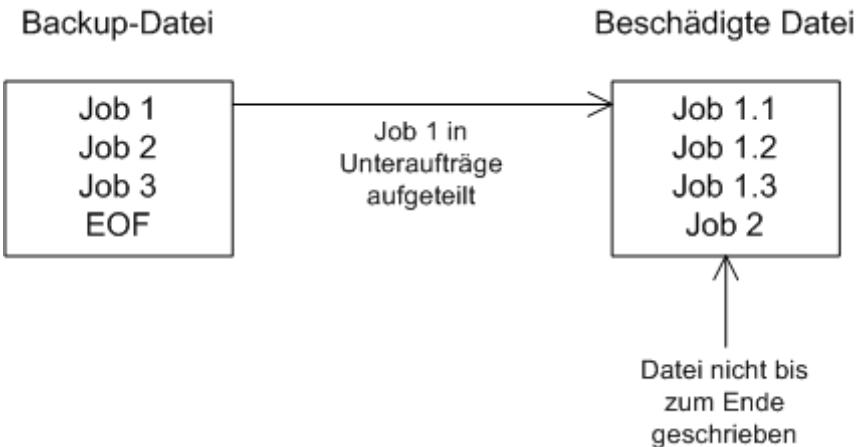


Abbildung 13.6: Vorgehen, wenn Datei beschädigt ist

13.1.3 Logging

Die Klasse *FDZLogging* baut auf der Klasse *FDZFileIO* auf und zielt darauf ab Dateizugriffe speziell für Protokolldateien anzubieten.

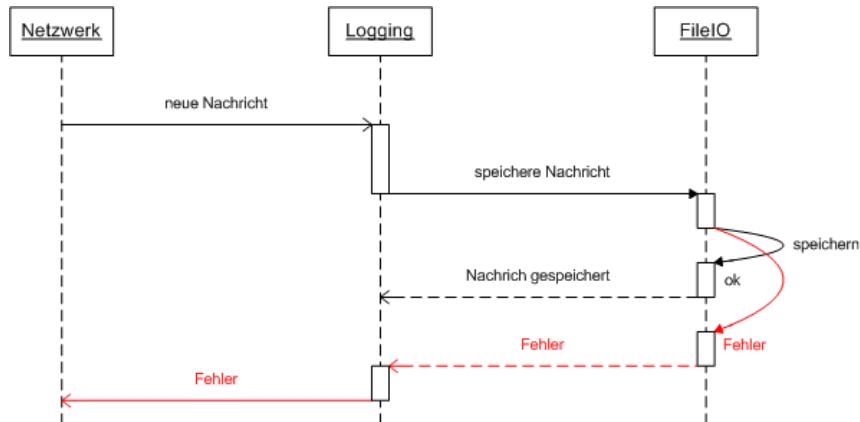


Abbildung 13.7: Schreiben in die LogDatei

Die Klasse *FDZLogging* bietet die zwei statischen Methoden *getInstance()* und *getInstance(string)* an. Da diese Klasse nur ein einziges Mal instanziert werden soll, ist ihr Konstruktor nicht öffentlich zugänglich und es muss über eine der beiden oben genannten Funktionen auf sie zugegriffen werden. *getInstance(string)* wird beim ersten Aufruf verwendet, da in dieser Funktion die programmweite Logging-Datei festgelegt wird. Wird diese Funktion später noch einmal aufgerufen, wird der angegebene Dateiname einfach ignoriert und eine Referenz auf das bereits bestehende Objekt zurückgeliefert. Die Variante *getInstance()* dagegen liefert, solange noch keine Instanz existiert Null zurück, ansonsten das einzige Objekt von *FDZLogging*.

13.1 Gemeinsame Klassen

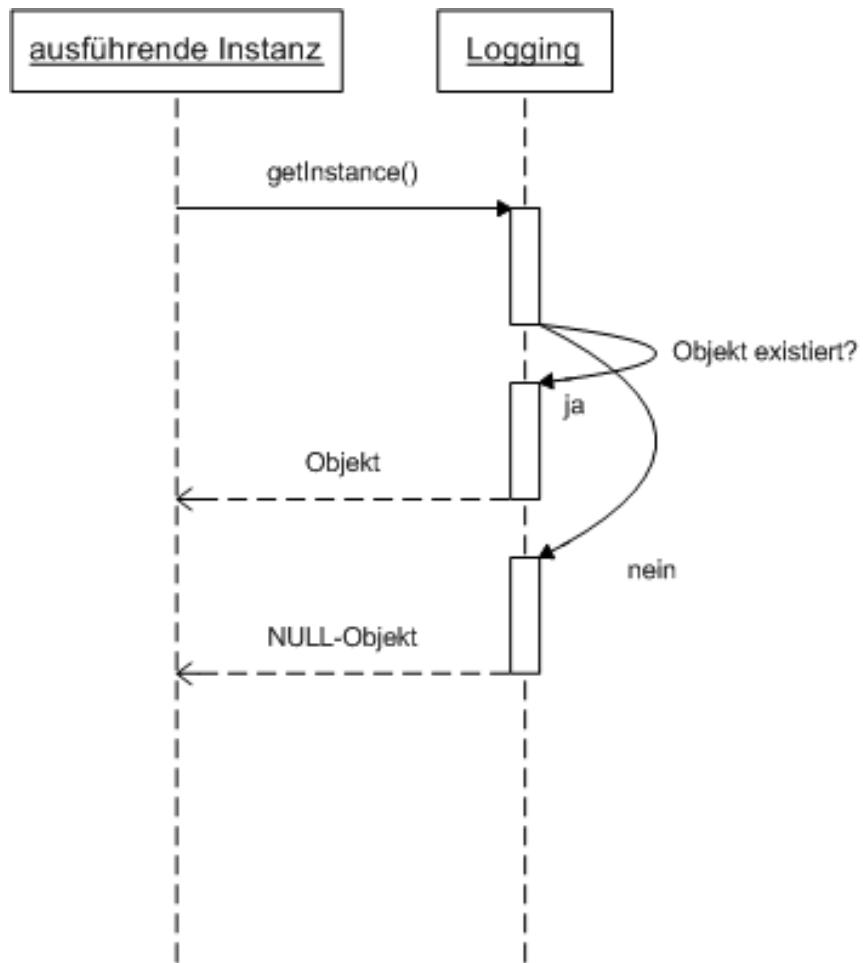


Abbildung 13.8: aktuelle Instanz der Logging-Klasse anfordern

Mit der Funktion `writeMsg(string)` wird an die Logdatei eine Zeile angehängt. Diese kann mit `getLastMsg()` wieder ausgelesen werden.

13.2 Modul Control

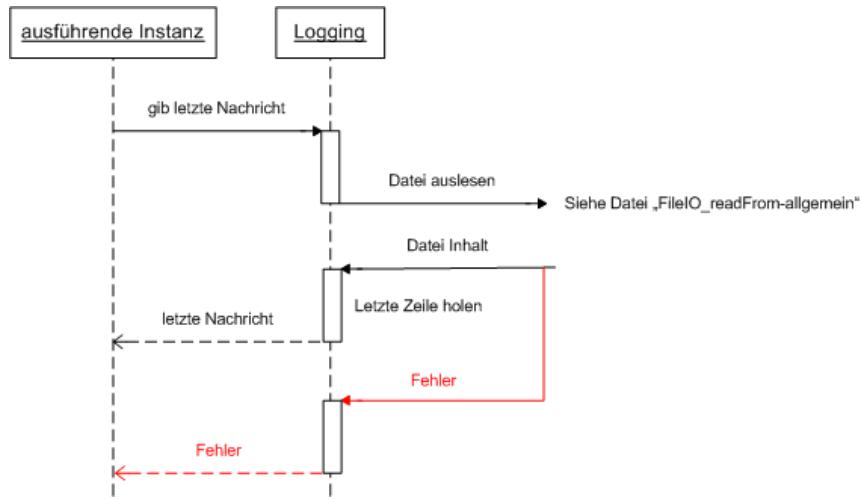


Abbildung 13.9: letzte Nachricht in der Log-Datei auslesen

13.2 Modul Control

13.2.1 Konstruktion der Steuerung

13.2.1.1 Textuelle Beschreibung der Steuerung

Die Steuerung bildet die Verknüpfung zwischen den Modulen ?? JSAP, Roboter, Lager, Band und E/A Station. Es wird der logische Ablauf eines Bestückungsauftrages und einer Bestandsabfrage durchgeführt. Hierzu werden die Kommandos von JSAP über das Netzwerk empfangen, in Teilaufträge unterteilt und zu den Subsystemen gesendet.

13.2.1.1.1 Textuelle Beschreibung einer Bestandsabfrage

Bei einer Bestandsabfrage wird wie in ?? beschrieben für jede Farbe ein Kommando zur Bestimmung des aktuellen Lagerbestandes erzeugt. Diese Nachrichten werden in einer Queue ?? abgelegt und durch den QueueTrustee zum Lager gesendet. Anschließend wartet die Steuerung auf die Antworten der Lagersteuerung. Ist die Bestandsabfrage an der Lagersteuerung beendet, wird eine Queue mit der Antwort an das JSAP System erzeugt und mit dem QueueTrustee versendet.

13.2.1.1.2 Textuelle Beschreibung eines Bestückungsauftrages

Bei einem Bestückungsauftrag wird wie in ?? beschrieben ein Kommando des JSAP zur Bestückung einer Produktpalette durchgeführt. Zunächst wird im Lager geprüft ob ein ausreichender Bestand zur Verfügung steht. Im Positiven Falle wird der Auftrag ausgeführt, und im negativen Falle eine Meldung an das JSAP gesendet, dass der Auftrag nicht produzierbar ist.

13.2 Modul Control

Für die Abarbeitung des Auftrages werden mehrere Queues ?? mit Befehlen an die Submodule erzeugt, die durch den QueueTrustee ?? zu den richtigen Modulen weitergeleitet wird. Ist die Bestückung erfolgreich durchgeführt worden, wird eine Erfolgsmeldung an das JSAP gesendet.

13.2.1.2 Klassendiagramm

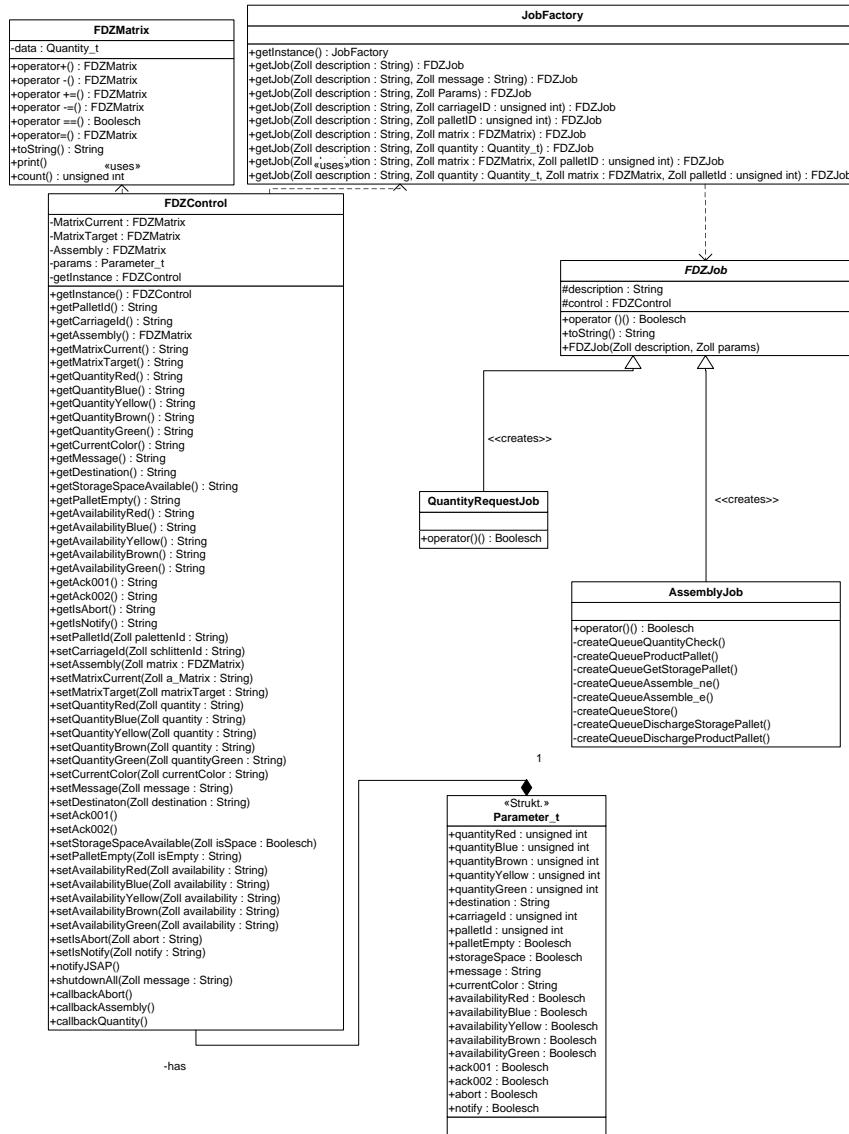


Abbildung 13.10: Klassendiagramm Control

Beschreibung siehe API

13.2 Modul Control

13.2.1.3 Hauptablauf

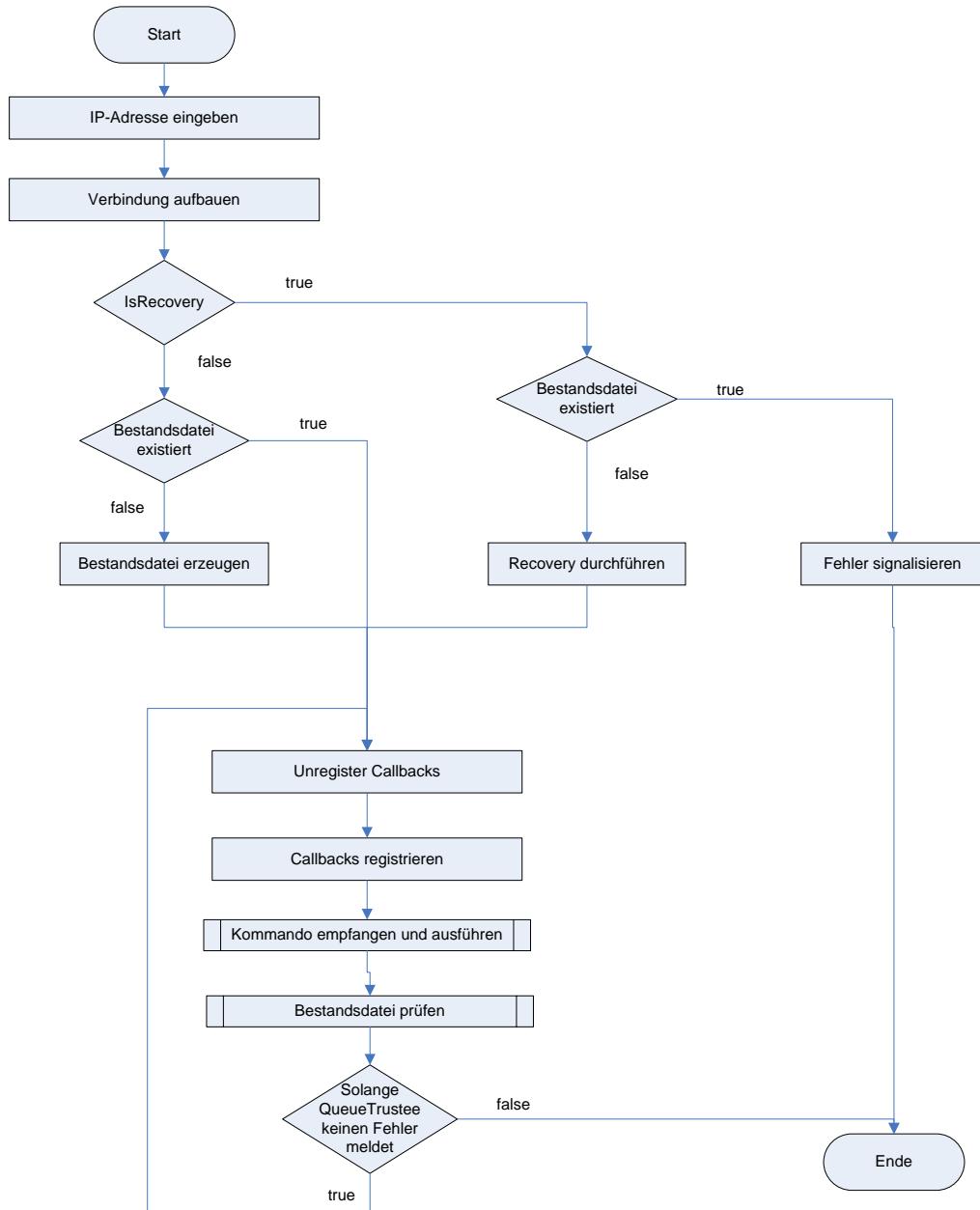


Abbildung 13.11: Hauptablauf Control

Beschreibung siehe API

13.2 Modul Control

13.2.1.3.1 FDZControl

13.2.2 Message Sequenz Charts fuer die Callbacks

13.2.2.1 Callback Quantity

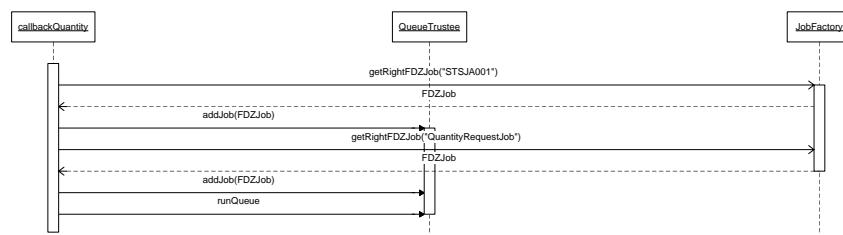


Abbildung 13.12: Callback Quantity

Wird das Kommando für eine Bestandsabfrage empfangen, springt das System in den Callback Quantity. Hier wird zunächst der Empfang der Nachricht positiv an das JSAP quittiert. Dann wird aus dem Lager der aktuelle Bestand an Smarties aller Farben abgefragt und dieser letztlich dem JSAP mitgeteilt.

Weitere Beschreibung siehe API

13.2 Modul Control

13.2.2.1.1 Callback Assembly

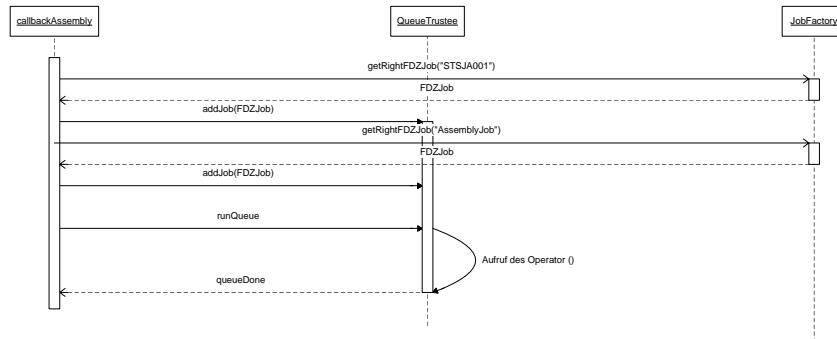


Abbildung 13.13: Callback Assembly

Hauptaufgabe von Control. Führt den von JSAP empfangenen Bestückungsjob durch. Weitere Beschreibung siehe API

13.2.2.1.2 Callback Abort

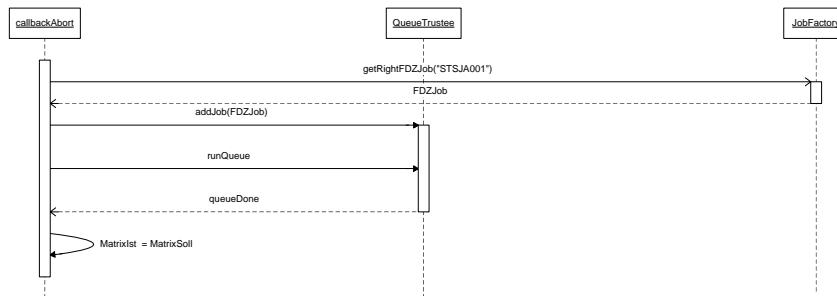


Abbildung 13.14: Callback Abort

Wird aufgerufen wenn das AbbruchKommando eines Auftrages vom JSAP empfangen wird. Zunächst wird dem JSAP der Empfang des Kommandos bestätigt, dannach alle Module in die Ausgangsposition zurückgebracht und dem JSAP der Erfolg gemeldet. Weitere Beschreibung siehe API

13.2 Modul Control

13.2.2.2 Programm - Ablauf - Plan AssemblyJob

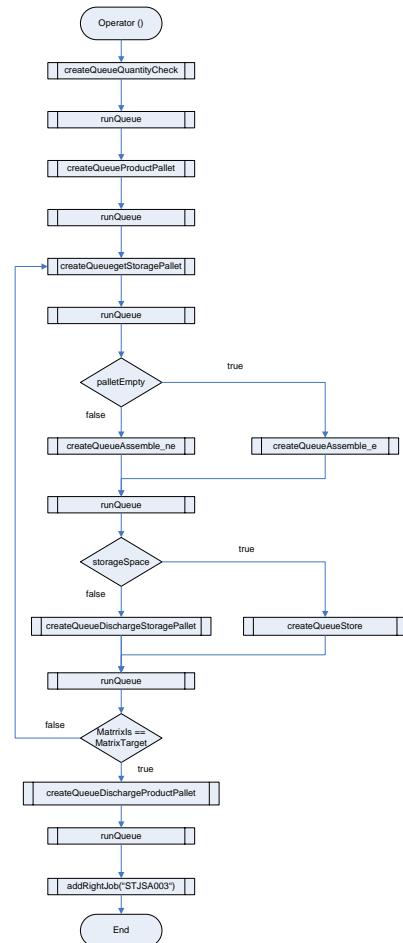


Abbildung 13.15: AssemblyJob

13.2.2.3 Beschreibung AssemblyJob

Hauptjob von Control. Hier wird die Queue fuer einen Bestueckungsjob erstellt Beschreibung siehe API

13.2 Modul Control

13.2.2.4 Message Sequenz Chart QuantityRequestJob

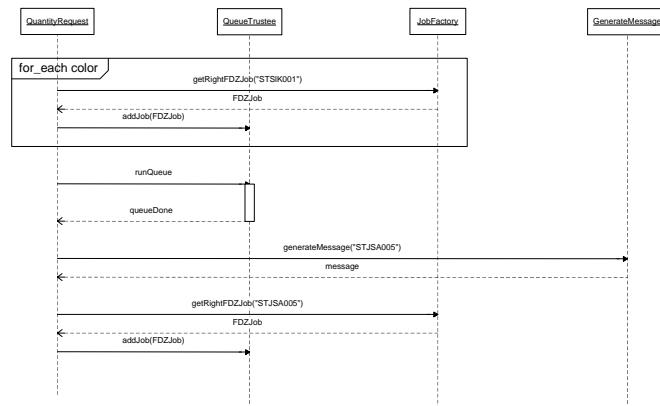


Abbildung 13.16: QuantityRequestJob

13.2.2.5 Beschreibung QuantityRequestJob

Führt die Abfrage des aktuellen Bestandes im Lager durch. Weitere Beschreibung siehe API

13.2 Modul Control

13.2.2.6 Programm - Ablauf - Plan fr die Queues

Beschreibung siehe API

13.2.2.6.1 Programm - Ablauf - Plan Queue Quantity Check

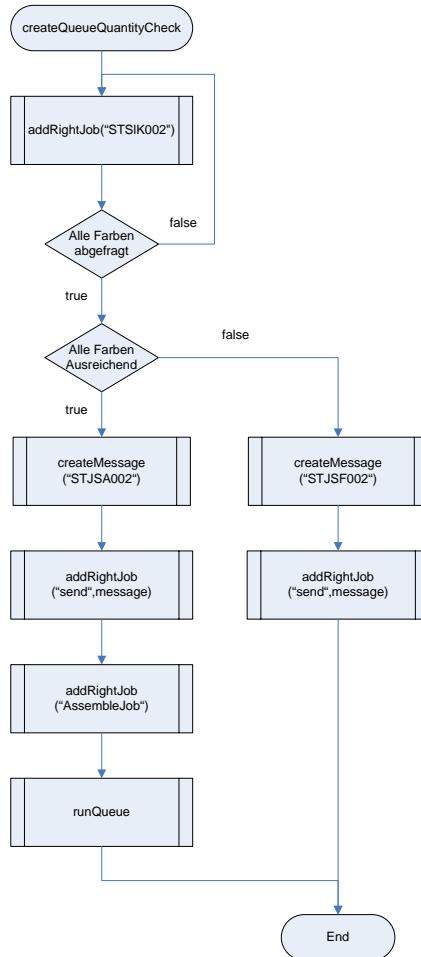


Abbildung 13.17: Queue Quantity Check

Hier werden nacheinander die Jobs erstellt, die den Bestand der einzelnen Farben abfragen. Lagermodul setzt die zugehoerigen Variablen in FDZControl. Weitere Beschreibung siehe API

13.2 Modul Control

13.2.2.6.2 Programm - Ablauf - Plan Queue Product Pallet

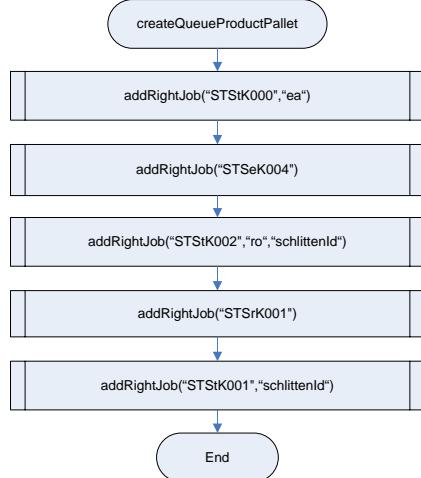


Abbildung 13.18: Queue Product Pallet

Hier werden die Jobs erstellt, die eine Produktpalette von der E/A-Station zum Roboter bringen und anschliesend den Schlitten freigeben. Weitere Beschreibung siehe API

13.2.2.6.3 Programm - Ablauf - Plan Queue Get Storage Pallet

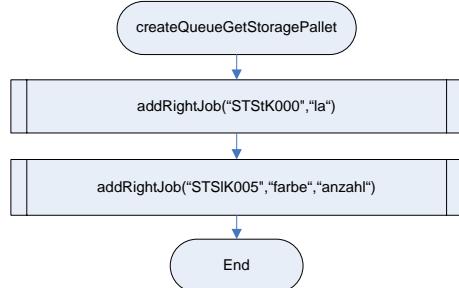


Abbildung 13.19: Queue Get Storage Pallet

Hier werden die Jobs erstellt, die einen Schlitten am Lager anfordert und eine Lagerpalette am Lager auslagert. Das Lagermodul setzt `palletEmpty` in `FDZControl` abhaengig davon, ob die Lagerpalette beim folgenden Bestueckungsv organg leer wird. `palletEmpty` beeinflusst im weiteren Ablauf des Vorganges, ob die LagerPalette zurueck ins Lager gefahren wird, oder ob sie zur E/A-Station zum Auffuellen gefahren wird. Weitere Beschreibung siehe API

13.2 Modul Control

13.2.2.6.4 Programm - Ablauf - Plan Queue Assembly not empty

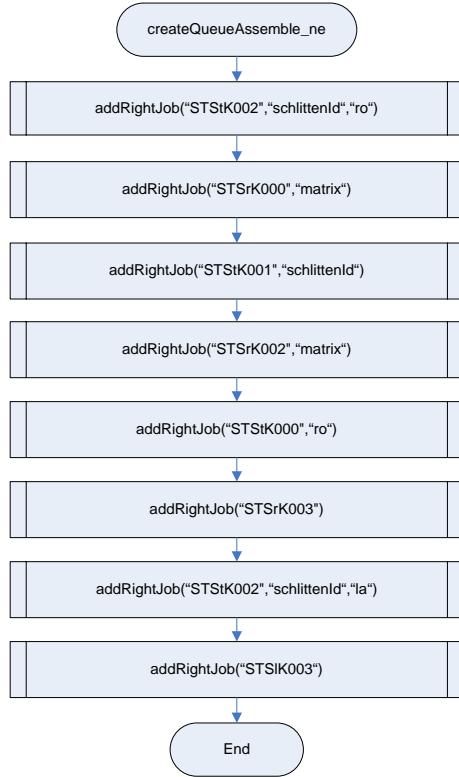


Abbildung 13.20: Queue Assembly not empty

Hier werden die Jobs erstellt, die folgende Aktionen durchfuehren:

1. Einen bestimmten Schlitten mit schlittenID am Roboter positionieren
2. Roboter die Lagerpalette auf Lagerplatz am Roboter heben lassen
3. Schlitten freigeben
4. Bestueckung durchfuehren
5. Schlitten am Roboter anfordern
6. Lagerpalette auf Schlitten heben
7. Lagerpalette am Lager positionieren
8. Lager fragen ob Platz ist

Die Lagerpalette ist wie in `createQueueGetStoragePallet` festgestellt wurde, NICHT LEER. Weitere Beschreibung siehe API

13.2 Modul Control

13.2.2.6.5 Programm - Ablauf - Plan Queue Assembly empty

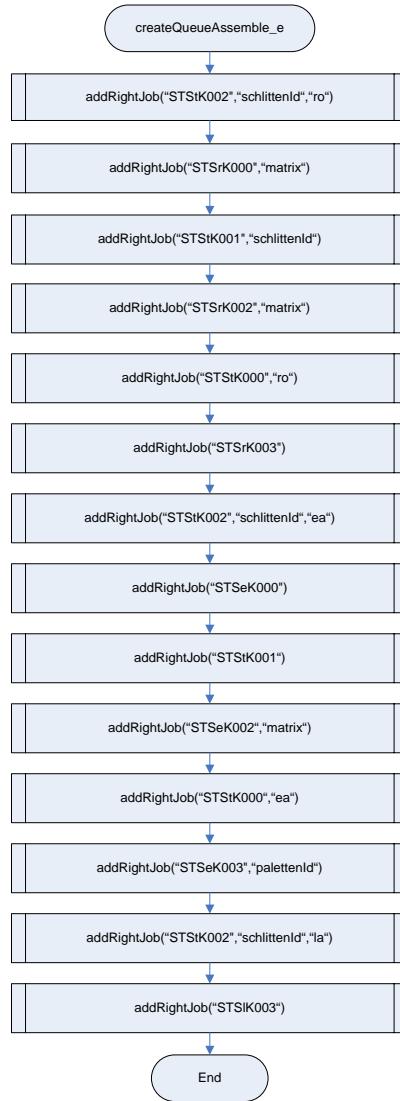


Abbildung 13.21: Queue Assembly empty

Hier werden die Jobs erstellt, die folgende Aktionen durchfuehren:

1. Einen bestimmten Schlitten mit schlittenID am Roboter positionieren
2. Roboter die Lagerpalette auf Lagerplatz am Roboter heben lassen
3. Schlitten freigeben
4. Bestueckung durchfuehren
5. Schlitten am Roboter anfordern
6. Lagerpalette auf Schlitten heben

13.2 Modul Control

7. Einen bestimmten Schlitten mit schlittenID an E/A positionieren
8. Leere Lagerpalette ausschleusen
9. Schlitten freigeben
10. Lagerpalette an E/A mit Matrix fuellen
11. Schlitten an E/A anfordern
12. Palette mit PalettenID an E/A einschleusen
13. Schlitten mit SchlittenID am Lager positionieren
14. Anfrage an Lager, ob es mindestens einen freien Stellplatz gibt

Die Lagerpalette ist wie in `createQueueGetStoragePallet` festgestellt wurde, LEER. Weitere Beschreibung siehe API

13.2.2.6.6 Programm - Ablauf - Plan Queue Discharge Storage Pallet

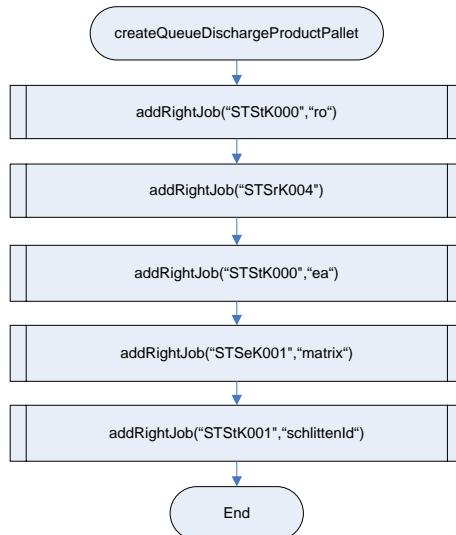


Abbildung 13.22: Queue Discharge Storage Pallet

Hier werden die Jobs erstellt, die folgende Aktionen durchfuehren:

1. Einen Schlitten mit SchlittenID an E/A positionieren
2. Lagerpalette ausschleusen
3. Schlitten freigeben

Weitere Beschreibung siehe API

13.2 Modul Control

13.2.2.6.7 Programm - Ablauf - Plan Queue Store

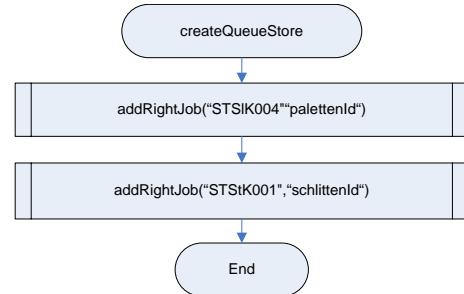


Abbildung 13.23: Queue Store

Hier werden die Jobs erstellt, die folgende Aktionen durchföhren:

1. Lagerpalette mit PalettenId um Lager einlagern
2. Schlitten freigeben

Diese Queue wird ausgeführt, wenn im Lager MINDESTENS EIN freier Platz zur Verfüigung steht.
Weitere Beschreibung siehe API

13.3 Modul Lager

13.2.2.6.8 Programm - Ablauf - Plan Queue Discharge Product Pallet

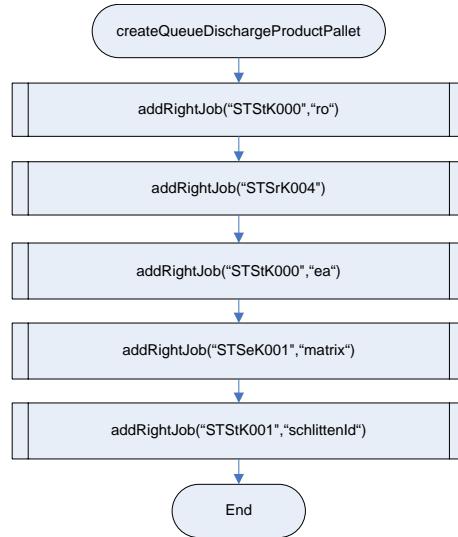


Abbildung 13.24: Queue DischargeProductPallet

Hier werden die Jobs erstellt, die folgende Aktionen durchfuehren:

1. Schlitten am Roboter anfordern
2. Produktpalette auf Band heben
3. Schlitten zu E/A fahren
4. Produktpalette mit MAtrix an E/A ausschleusen
5. Schlitten freigeben

Diese Queue wird ausgefuehrt, wenn matrixCurrent der matrixTarget entspricht, also der Bestueckungsauftrag komplett ausgefuehrt wurde oder der Befehl Abbruch von JSAP eingeleitet wurde. Weitere Beschreibung siehe API

13.3 Modul Lager

13.3.1 FDZStorageJob

13.3.1.1 Klassenbeschreibung

Diese Klasse repräsentiert die Funktionalität eines Jobs für das Submodul Lager-Steuerung.

13.3 Modul Lager

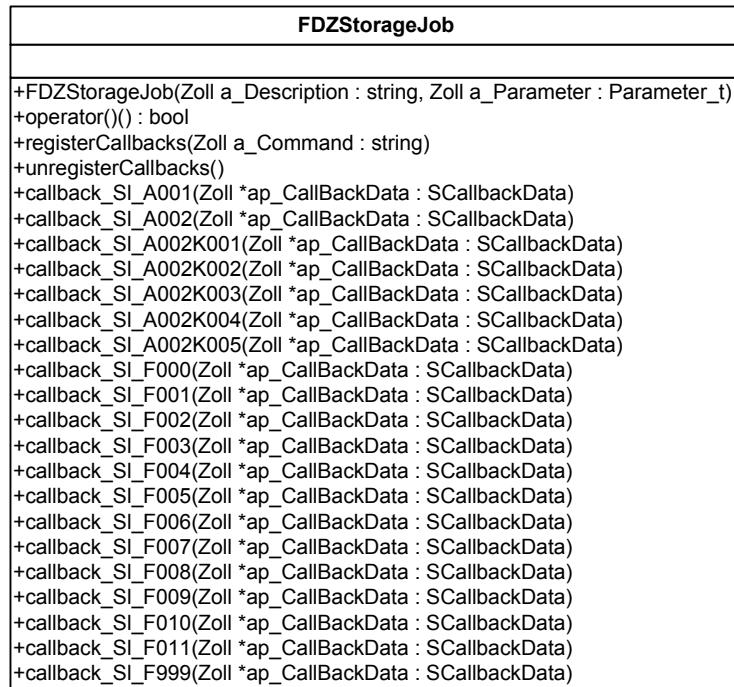


Abbildung 13.25: Klassendiagramm der Klasse FDZStorageJob

Die Klasse FDZStorageJob enthält einen Konstruktor, der als Übergabeparameter *a_Description* vom Typ *string* und *a_Parameter* vom Typ *Parameter_t* entgegennimmt. Sie stellt außerdem zwei Methoden zur Verfügung.

- *operator()()*: Der Klammeroperator wird überladen und hat einen Rückgabewert vom Typ *bool* und gibt somit wahr oder falsch zurück
- *registerCallbacks()*: Dient zur Registrierung aller Callbacks für einen bestimmten Befehl, welcher mittels des Übergabeparameters *a_Command* übergeben wird.
- *unregisterCallbacks()*: Dient zur Entfernung aller Callbacks für das Modul Lager.

Darüber hinaus können folgende 20 Callbacks registriert werden, welchen immer ein Zeiger auf die rufende Funktion übergeben wird.

- *callback_SI_A001()*: Befehl erhalten und kann abgearbeitet werden
- *callback_SI_A002()*: Befehl wurde ausgeführt
- *callback_SI_A002K001()*: Anzahl des Bestands einer Farbe
- *callback_SI_A002K002()*: Ausreichender Bestand vorhanden
- *callback_SI_A002K003()*: Anzahl der freien Stellplätze
- *callback_SI_A002K004()*: Auftrag erfolgreich ausgeführt

13.3 Modul Lager

- callback_SI_A002K005(): Auftrag erfolgreich ausgeführt, Lagerpalettennummer und die Belegung der Smarties auf der Palette
- callback_SI_F000(): Unbekannter Befehl / Unbekannte Befehlssyntax
- callback_SI_F001(): Fehler bei lesendem und/oder schreibenden Zugriff auf Datenbasis
- callback_SI_F002(): Bestand reicht nicht aus; Anzahl im Lager befindlichen Smarties der angeforderten Farbe
- callback_SI_F003(): Farbe ungültig
- callback_SI_F004(): Anzahl ungültig
- callback_SI_F005(): Palette konnte nicht eingelagert werden, Matrix inhomogen
- callback_SI_F006(): Palette konnte nicht eingelagert werden, kein Lagerplatz frei
- callback_SI_F007(): Palette konnte nicht eingelagert werden, Lagerpalette leer
- callback_SI_F008(): Palette konnte nicht eingelagert werden, mechanischer Fehler beim Einlagern
- callback_SI_F009(): Palette konnte nicht eingelagert werden, Palleten-ID befindet sich bereits im Lager
- callback_SI_F010(): Palette konnte nicht ausgelagert werden, nicht genügend Smarties im Lager
- callback_SI_F011(): Palette konnte nicht ausgelagert werden, mechanischer Fehler beim Auslagern
- callback_SI_F999(): Recovery des Systems

13.3.1.2 Ablaufbeschreibung

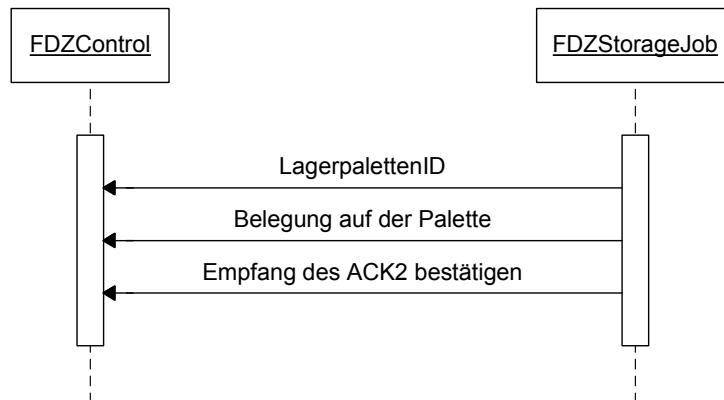


Abbildung 13.26: Sequenzdiagramm des Callbacks callback_SI_A002K005

13.4 Modul Transport

Wie im Diagramm zu sehen ist, wird beim Aufrufen des Callbacks `callback_SI_A002K005` die PalettenID und die Belegung der Palette an Control übergeben, sowie der Empfang des ACK2 bestätigt.

13.4 Modul Transport

13.4.1 FDZTransportJob

13.4.1.1 Klassenbeschreibung

Diese Klasse repräsentiert die Funktionalität eines Jobs für das Submodul Band-Steuerung.

FDZTransportJob
+FDZTransportJob(Zoll a_Description : string, Zoll a_Parameter : Parameter_t) +operator()() : bool +registerCallbacks(Zoll a_Command : string) +unregisterCallbacks() +callback_St_A001(Zoll *ap_CallBackData : SCallbackData) +callback_St_A002(Zoll *ap_CallBackData : SCallbackData) +callback_St_A002K001(Zoll *ap_CallBackData : SCallbackData) +callback_St_F000(Zoll *ap_CallBackData : SCallbackData) +callback_St_F001(Zoll *ap_CallBackData : SCallbackData) +callback_St_F002(Zoll *ap_CallBackData : SCallbackData) +callback_St_F003(Zoll *ap_CallBackData : SCallbackData) +callback_St_F999(Zoll *ap_CallBackData : SCallbackData)

Abbildung 13.27: Klassendiagramm der Klasse FDZTransportJob

Die Klasse FDZTransportJob enthält einen Konstruktor, der als Übergabeparameter `a_Description` vom Typ `string` und `a_Parameter` vom Typ `Parameter_t` entgegennimmt. Sie stellt außerdem zwei Methoden zur Verfügung.

- `operator()()`: Der Klammeroperator wird überladen und hat einen Rückgabewert vom Typ `bool` und gibt somit wahr oder falsch zurück
- `registerCallbacks()`: Dient zur Registrierung aller Callbacks für einen bestimmten Befehl, welcher mittels des Übergabeparameters `a_Command` übergeben wird.
- `unregisterCallbacks()`: Dient zur Entfernung aller Callbacks für das Modul Band.

Darüber hinaus können folgende acht Callbacks registriert werden, welchen immer ein Zeiger auf die rufende Funktion übergeben wird.

- `callback_St_A001()`: Befehl erhalten und kann abgearbeitet werden
- `callback_St_A002()`: Auftrag erfolgreich ausgeführt
- `callback_St_A002K001()`: ID des freien Schlittens
- `callback_St_F000()`: Unbekannter Befehl / Unbekannte Befehlssyntax

13.5 Modul Roboter

- callback_St_F001(): Befehl wurde nicht ausgeführt
- callback_St_F002(): Schlitten ist nicht angekommen
- callback_St_F003(): Allgemeiner Hardware-Fehler
- callback_St_F999(): Transportsystem kann nicht weiterarbeiten

13.4.1.2 Ablaufbeschreibung



Abbildung 13.28: Sequenzdiagramm des Callbacks `callback_St_F002`

Wie im Diagramm zu sehen ist, wird beim Aufrufen des Callbacks `callback_St_F002` die SchlittenID an Control übergeben, sowie Befehl zum herunterfahren aller Subsysteme ausgelöst, mit der Begründung, der Schlitten ist nicht angekommen.

13.5 Modul Roboter

13.5.1 FDZRobotJob

13.5.1.1 Klassenbeschreibung

Diese Klasse repräsentiert die Funktionalität eines Jobs für das Submodul Roboter-Steuerung.

13.5 Modul Roboter

FDZRobotJob
<pre>+FDZRobotJob(Zoll a_Description : string, Zoll a_Parameter : Parameter_t) +operator()() : bool +registerCallbacks(Zoll a_Command : string) +unregisterCallbacks() +callback_Sr_A001(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_A002(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_A002K002(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_A002K003_K005(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_F000(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_F001(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_F002(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_F003(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_F004(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_F005(Zoll *ap_CallBackData : SCallbackData) +callback_Sr_F999(Zoll *ap_CallBackData : SCallbackData)</pre>

Abbildung 13.29: Klassendiagramm der Klasse FDZRobotJob

Die Klasse FDZRobotJob enthält einen Konstruktor, der als Übergabeparameter *a_Description* vom Typ *string* und *a_Parameter* vom Typ *Parameter_t* entgegennimmt. Sie stellt außerdem zwei Methoden zur Verfügung.

- *operator()()*: Der Klammeroperator wird überladen und hat einen Rückgabewert vom Typ *bool* und gibt somit wahr oder falsch zurück
- *registerCallbacks()*: Dient zur Registrierung aller Callbacks für einen bestimmten Befehl, welcher mittels des Übergabeparameters *a_Command* übergeben wird.
- *unregisterCallbacks()*: Dient zur Entfernung aller Callbacks für das Modul Roboter.

Darüber hinaus können folgende elf Callbacks registriert werden, welchen immer ein Zeiger auf die rufende Funktion übergeben wird.

- *callback_Sr_A001()*: Der Befehl wurde empfangen und konnte interpretiert werden
- *callback_Sr_A002()*: Befehl wurde verstanden
- *callback_Sr_A002K002()*: Befehl wurde verstanden
- *callback_Sr_A002K003_K005()*: Befehl wurde verstanden
- *callback_Sr_F000()*: Befehl wurde nicht verstanden
- *callback_Sr_F001()*: Befehl wurde nicht ausgeführt
- *callback_Sr_F002K002_K003_K005()*: Keine Lagerpalette auf A/B vorhanden
- *callback_Sr_F003K000_K002_K004()*: Keine Produktpalette auf X vorhanden
- *callback_Sr_F004K000_K005()*: Plätze A/B sind bereits belegt
- *callback_Sr_F005K001()*: Befehl wurde nicht ausgeführt
- *callback_Sr_F999()*: Fehler nach Recovery/Neustart

13.6 Modul E/A-Station

13.5.1.2 Ablaufbeschreibung

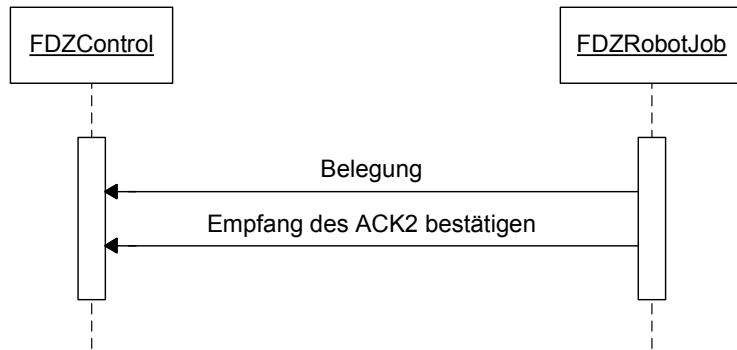


Abbildung 13.30: Sequenzdiagramm des Callbacks `callback_Sr_A002K003_K005`

Wie im Diagramm zu sehen ist, wird beim Aufrufen des Callbacks `callback_Sr_A002K003_K005` die Belegung an Control übergeben, sowie der Empfang des ACK2 bestätigt.

13.6 Modul E/A-Station

13.6.1 FDZIOJob

13.6.1.1 Klassenbeschreibung

Diese Klasse repräsentiert die Funktionalität eines Jobs für das Submodul E/A-Steuerung.

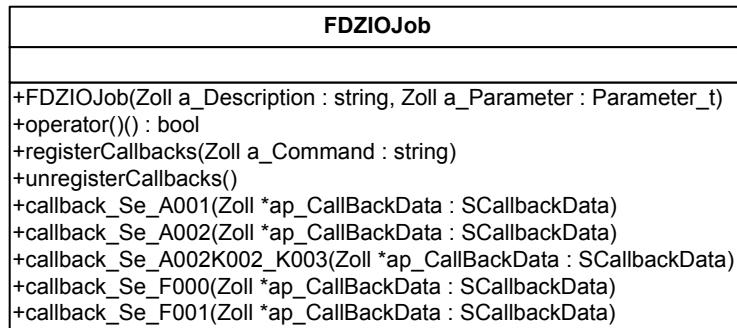


Abbildung 13.31: Klassendiagramm der Klasse FDZIOJob

Die Klasse FDZIOJob enthält einen Konstruktor, der als Übergabeparameter `a_Description` vom Typ `string` und `a_Parameter` vom Typ `Parameter_t` entgegennimmt. Sie stellt außerdem zwei Methoden zur Verfügung.

13.6 Modul E/A-Station

- `operator()()`: Der Klammeroperator wird überladen und hat einen Rückgabewert vom Typ `bool` und gibt somit wahr oder falsch zurück
- `registerCallbacks()`: Dient zur Registrierung aller Callbacks für einen bestimmten Befehl, welcher mittels des Übergabeparameters `a_Command` übergeben wird.
- `unregisterCallbacks()`: Dient zur Entfernung aller Callbacks für das Modul E/A-Steuerung.

Darüber hinaus können folgende fünf Callbacks registriert werden, welchen immer ein Zeiger auf die rufende Funktion übergeben wird.

- `callback_Se_A001()`: Befehl wurde verstanden
- `callback_Se_A002()`: Befehl wurde ausgeführt
- `callback_Se_A002K002_K003()`: ID der befüllten Lagerpalette, sowie ihre tatsächliche Bestückung zurückgeben
- `callback_Se_F000()`: Befehl wurde nicht verstanden
- `callback_Se_F001()`: Befehl wurde nicht ausgeführt

13.6.1.2 Ablaufbeschreibung

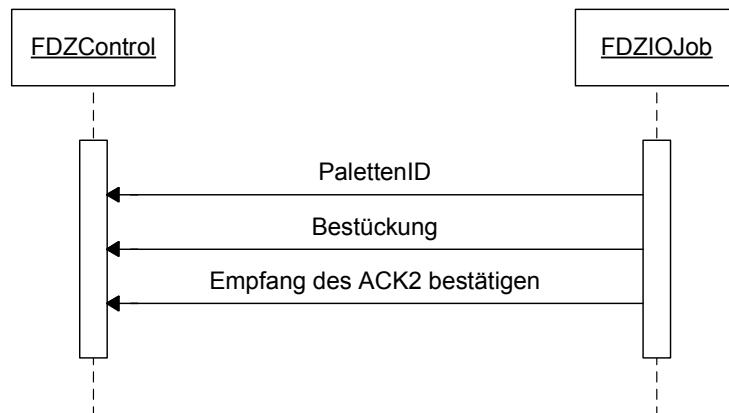


Abbildung 13.32: Sequenzdiagramm des Callbacks `callback_Se_A002K002_K003`

Wie im Diagramm zu sehen ist, wird beim Aufrufen des Callbacks `callback_Se_A002K002_K003` die PalettenID, die Bestückung an Control übergeben, sowie der Empfang des ACK2 bestätigt.

13.7 Fehlerfälle

13.7.1 FDZException

13.7.1.1 Klassenbeschreibung

Die Klasse FDZException wurde entworfen um auftretende Fehlerfälle abzufangen und diese mit einer aussagekräftigen Beschreibung an Control hoch zureichen.

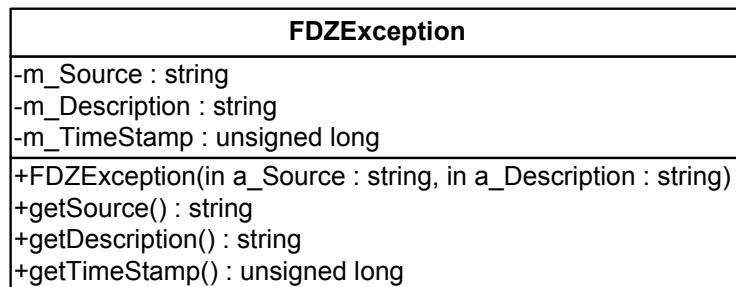


Abbildung 13.33: Klassendiagramm der Klasse FDZException

Die Klasse FDZException enthält drei private Variablen.

- m_Source vom Typ *string* hält die Quelle der Exception
- m_Description vom Typ *string* enthält eine genauere Beschreibung der Ausnahme
- m_TimeStamp vom Typ *unsigned long* stellt den Zeitstempel des Fehlers bereit

Des weiteren hat die Klasse einen Konstruktor, der als Übergabeparameter *a.Source* und *a.Description* vom Typ *string* entgegennimmt. Sie stellt außerdem drei Methoden zur Verfügung.

- getSource() gibt einen *string* zurück mit der Quelle der Ausnahme
- getDescription() gibt die Beschreibung als *string* zurück
- getTimeStamp() gibt den Zeitstempel der Exception als *unsigned long* zurück

13.7.1.2 Anwendungsbeschreibung

Die FDZException stellt dem Programmierer die gewohnten Befehle, wie try, catch und throw, zur Verfügung, jedoch mit FdZ spezifischer Funktionalität. Um die FDZException nützen zu können muss die *fdz_exception.hpp* in den Quellcode inkludiert werden. Danach kann wie gewohnt ein try-catch-Block aufgebaut werden, wie folgende Beispiele zeigen.

13.8 Modul Recovery

```
int foo(int bar)
{
    if (bar == 0)
    {
        // Throw exception because bar is zero
        throw FDZException("foo", "bar is zero.");
    }
    else
    {
        // Return value of foo()
        return bar;
    }
}

try
{
    // Do some foo()
    x = foo(5);
}
catch (FDZException exp)
{
    // Information for the thrown exception
    cout << "Source: " << exp.getSource() << endl;
    cout << "Description: " << exp.getDescription() << endl;
    cout << "TimeStamp: " << exp.getTimeStamp() << endl;
}
```

Wie im Beispiel ersichtlich, wird in der foo()-Methode der Funktionsname als Quelle angeben, sowie eine detaillierte Fehlerbeschreibung. Außerdem wird ein eindeutiger Zeitstempel bei dem Auftreten der Exception erzeugt. Alle Daten lassen sich wie zu erkennen ist aus dem exp-Objekt mit den entsprechenden Methoden auslesen.

13.8 Modul Recovery

13.8.1 JobFactory

Die JobFactory erstellt anhand einer Job-Bezeichnung, welche sich an die Kommandos der Steuerung anlehnt, entsprechende Job-Klassen. Diese Job-Klassen können der Queue hinzugefügt werden. Es können der JobFactory auch Parameter mitgegeben werden, die diese Jobs dann nutzen.

13.8 Modul Recovery

Job erzeugen (Parameter sind optional)

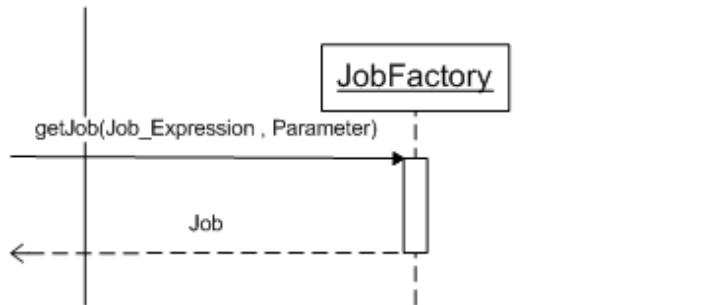


Abbildung 13.34: Erzeugen einer Job-Instanz

13.8.2 QueueTrustee

13.8.2.1 Jobs hinzufügen

Dem QueueTrustee können Jobs übergeben werden. Dieser entscheidet selbst ob die Jobs an der Anfang oder das Ende der Queue angehängt werden sollen.

Job hinzufügen

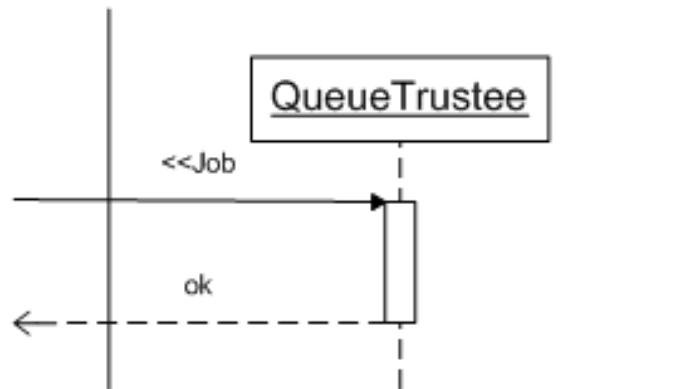


Abbildung 13.35: Hinzufügen eines Jobs zur Queue

13.8.2.2 Jobs abarbeiten

Der jeweils aktuell erste Job wird ausgeführt, danach wird dieser gelöscht und die Queue erneut gespeichert.

13.8 Modul Recovery

Job abarbeiten

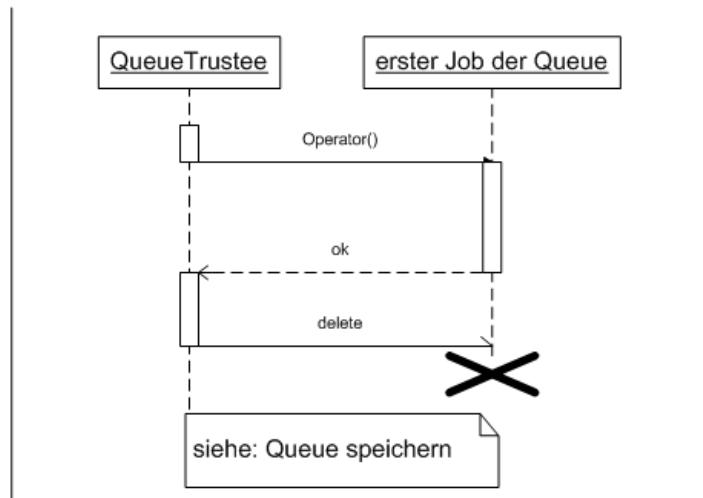


Abbildung 13.36: Ausführen eines Jobs aus der Queue

13.8.2.3 Queue abarbeiten

Es wird geprüft ob wir im Recoverfall sind, wenn führen wir Recovery durch, sonst wird die Queue gespeichert und jeder Job der Queue abgearbeitet.

13.8 Modul Recovery

Queue abarbeiten

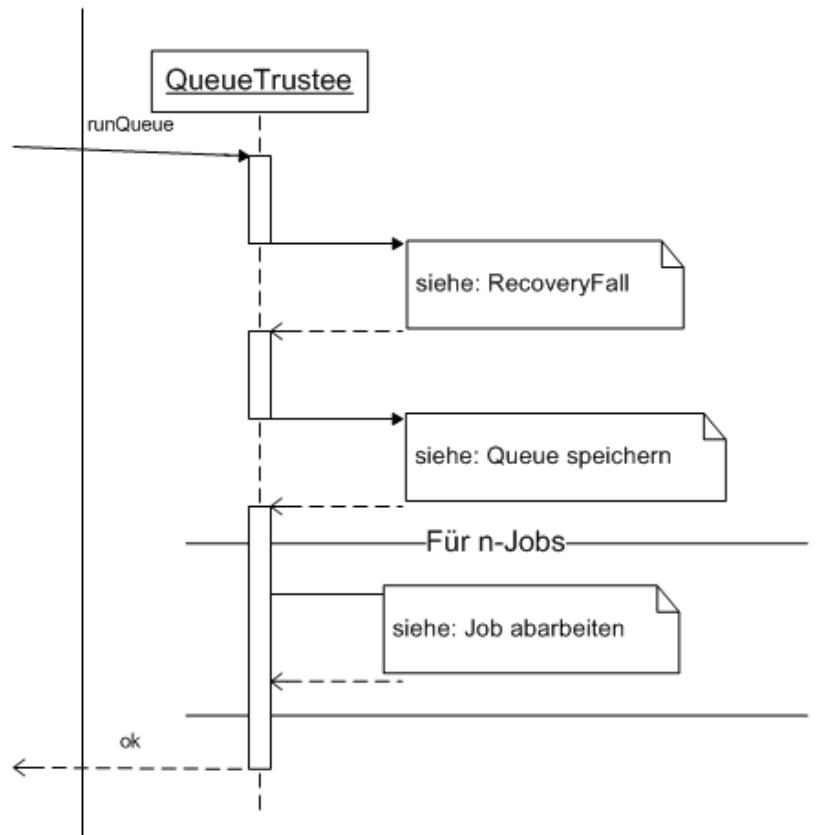


Abbildung 13.37: Abarbeiten der Jobs in der Queue

13.8.2.4 Queue speichern

Zunächst wird von Control der RestoreJob angefordert, dessen Aufgabe das Wiederherstellen der für den Auftrag wichtigen Informationen ist. Dieser wird an den Anfang der Queue-Datei gestellt wird. Alle Jobs der Queue werden gelesen und in die aktuelle Queue-Datei geschrieben.

13.8 Modul Recovery

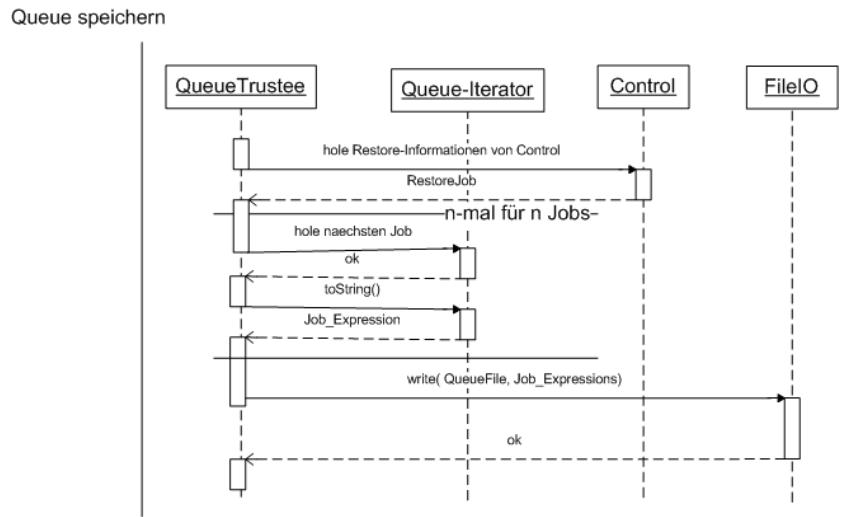


Abbildung 13.38: Speichern der Queue in einer Datei

13.8.3 Recovery

Als erstes wird geprüft, ob wir überhaupt im RecoveryFall sind, wenn dann wird die Recovery-Datei umbenannt und gelesen. Umbenannt werden muss diese, um ein Abstürzen im RecoveryFall „abzufangen“, denn die Recovery-Datei würde durch die neue Queue überschrieben werden. Die Jobs in der Recovery-Datei werden herausgeparst und jeweils mit der JobFactory in gültige Jobs umgewandelt, welche dann in die Queue eingefügt werden. Nun da alle Jobs wieder in der Queue sind -der erste Job ist der RestoreJob, der die Informationen von Control wieder herstellt- kann diese abgearbeitet werden.

13.8 Modul Recovery

RecoveryFall

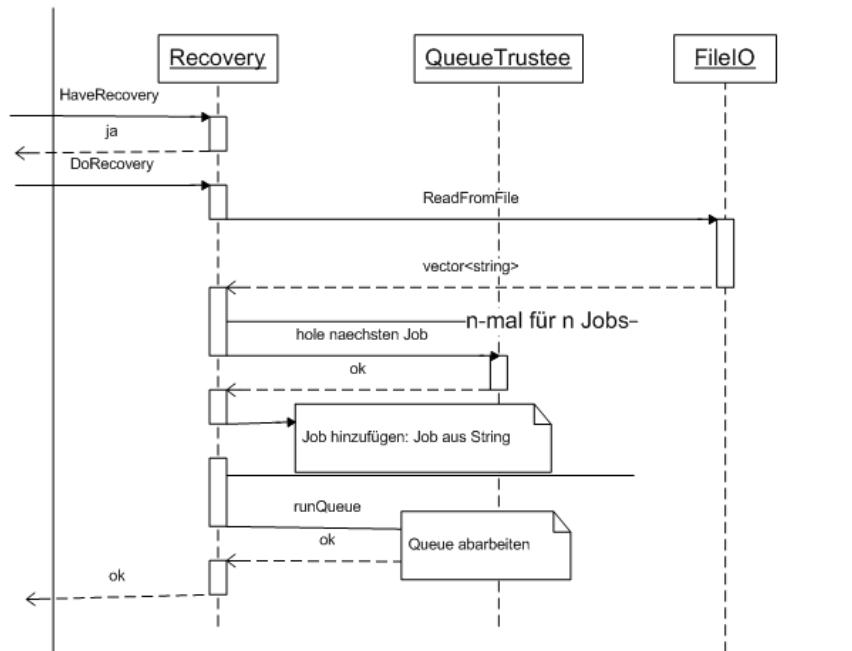


Abbildung 13.39: Durchführung des Recovery's

13.8 Modul Recovery

13.8.4 Die Klassen des Recovery's

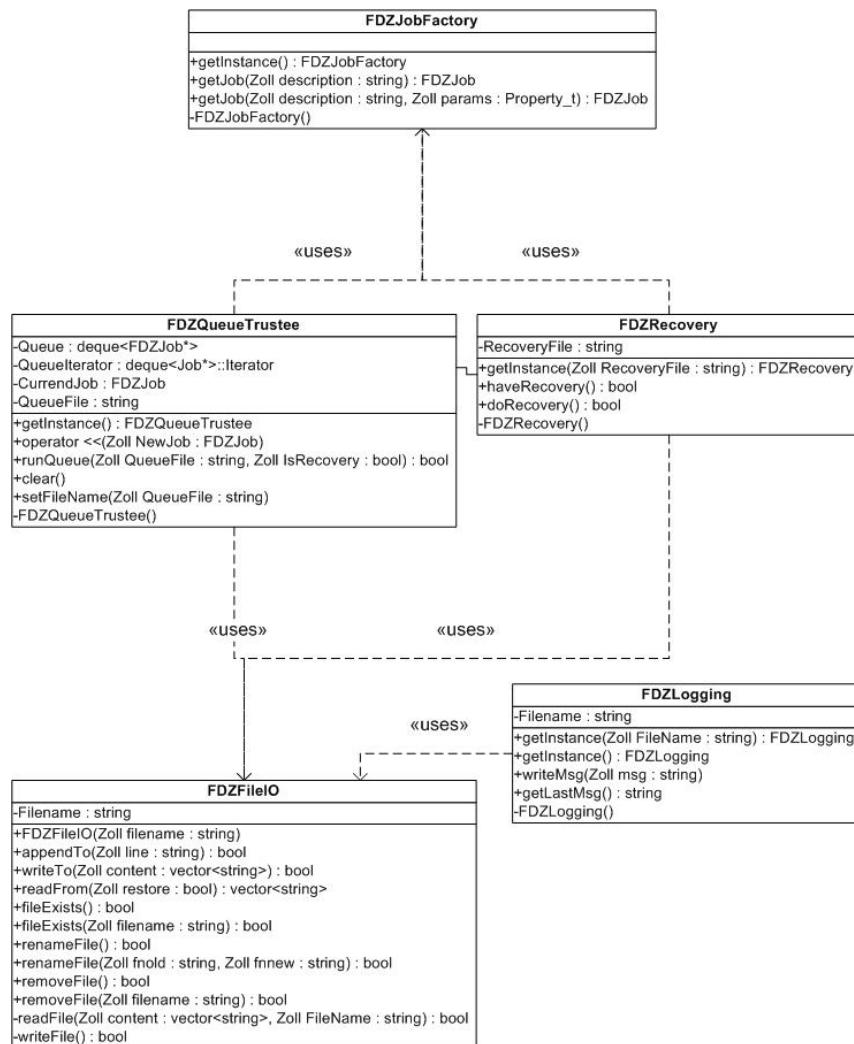


Abbildung 13.40: Klassendiagramm Recovery

Beschreibung siehe Doxygen.

14 Transportsystem

14.1 Aufbau / Struktur der Software

Die Software für das Bandsystem besteht im wesentlichen aus einem Teil, der auf der Siemens SPS des Transportbandes läuft und der Bandsteuerung, die mit den übergeordneten Modulen (Steuerung) kommuniziert. Zur Erleichterung der Kommunikation wird als Schnittstelle zwischen den beiden Komponenten nicht wie bei den anderen Modulen unmittelbar TCP/IP verwendet, sondern das SCADA-System ZenOn¹ als „Übersetzer“ dazwischengeschaltet.

Damit ergibt sich folgende Kommunikationsstruktur:

- Die Transportsteuerung spricht zenOn über TCP/IP an und sendet/empfängt Messages mit Befehlen.
- zenOn läuft parallel zur Transportsteuerung als Server und zeigt eine Visualisierung für den Zustand des Transportbandes an.
- Das Transportband selbst arbeitet autark und stellt die Weichen und die Stopper entsprechend den von ZenOn gesetzten Kommunikationsvariablen ein.

14.2 Kommunikation Transportsteuerung – zenOn – SPS

Zur „Verkehrsregelung“ sind noch folgende Regeln zu berücksichtigen:

- Es darf sich immer nur ein Schlitten im Bereich der Weiche aufhalten.
- Schlitten auf dem Seitenband haben Vorfahrt gegenüber dem Hauptband – d.h: wenn ein Schlitten auf dem Seitenband und ein Schlitten auf dem Hauptband gleichzeitig an der Weiche ankommen, muss der Schlitten auf dem Hauptband warten, bis der Schlitten auf dem Seitenband den Bereich der Weiche verlassen hat.²

¹<http://www.copadata.de/>

²So soll ein „Überlauf“ des Seitenbands verhindert werden, falls sich viele Schlitten im System befinden. Falls dennoch zu viele Schlitten vom Hauptband auf das Seiteband geleitet werden, trägt die Strategie nicht. Dieses Problem wird durch Begrenzung der Gesamtzahl der Träger vermieden.

14.3 Konstruktion des Transportsystems

Ablauf der Transportsteuerung: Erhält die Transportsteuerung eine Message von Control, wird diese in einzelne Teile zerlegt und eine Befehlskette erzeugt, die an Zenon gesendet wird. Nach jedem einzelnen Befehl der Befehlskette wird auf die Bestätigung gewartet, dass dieser ausgeführt wurde. Bevor der Nächste übermittelt wird, wird nochmals 5 Sekunden gewartet aufgrund der schlechten Performance der SPS bzw. Zenon.

14.3 Konstruktion des Transportsystems

14.3.1 Klassendiagramm Transportsteuerung



Abbildung 14.1: Klassendiagramm

Oben stehendes Klassendiagramm gibt eine Übersicht über alle verwendeten Klassen und deren Methoden. Eine genauere Beschreibung kann der Javadoc entnommen werden.

14.3.1.1 Sequenzdiagramm Transportsteuerung

Abbildung 14.2: Sequenzdiagramm

14.3 Konstruktion des Transportsystems

Oben stehendes Sequenzdiagramm beschreibt den grundlegenden Ablauf vom Starten der Applikation einschließlich Verbindungsauftbau über das Abarbeiten eines empfangenen Kommandos von Control bis hin zum Senden von Ack1/Ack2. Zunächst wird eine Instanz der Verarbeitung (Verarbeitung) angelegt, die grafische Oberfläche (DesktopView) erzeugt und anschließend die IP-Adresse von Control ermittelt. Im Falle eines Recovery wird dem Anwender eine Wiederherstellungsabfrage (WiederherstellungsAbfrage) angezeigt, auf der er den weiteren Ablauf des Programms entscheiden kann (Recovery ausführen oder verwerfen). Anschließend wartet die Anwendung auf eine Verbindung des Zenon-Clients und stellt daraufhin eine Verbindung zu Control her. Nun befindet sich die Applikation im Wartezustand (while-Schleife in TransportMain). Sobald eine Message von Control eintrifft, beginnt die eigentliche Verarbeitung: Zunächst wird ein Ack1 gesendet um das erfolgreiche Empfangen der Nachricht zu bestätigen. Die empfangene Message wird zerlegt und neue Messages für die unterhalb der Transportsteuerung angesiedelten Ebenen erzeugt (in Verarbeitung). Diese Messages werden einzeln versandt und bei jeder wird auf Ack1 und Ack2 gewartet ehe die Nächste bearbeitet wird. Sind alle Befehle erfolgreich ausgeführt worden, so wird ein Ack2 an Control übermittelt und der nächste Befehl kann empfangen und verarbeitet werden (while-Schleife in TransportMain).

14.3 Konstruktion des Transportsystems

Beschreibung der Recovery:

Funktionsprinzip:

Bei Absturz der Transportsteuerung (durch z.B. Not-Aus oder Stromausfall) soll das Programm genau an der Stelle fortsetzen, an der es abgestürzt ist. Optional kann aber auch der Initialzustand hergestellt werden. (Dies kann beim Starten nach einem Absturz ausgewählt werden).

Um eine lückenlose Fortsetzung des Programms nach einem Ausfall zu gewährleisten ist es notwendig, eine genaue Abarbeitungsliste persistent zu speichern. Diese Liste muss nach jedem einzelnen Schritt aktualisiert werden. Nach einem Absturz wird diese Liste eingelesen und abgearbeitet.

Genaue Erläuterung der Recovery:

Das Sicherungssystem baut auf der prinzipiellen Funktionsweise der Transportsteuerung auf:

- Erhalt eines Befehls vom Control
- Befehl verstanden: Schicke ACK1 an Control
- Befehl umwandeln (zum Teil aufteilen in mehrere Befehle) und an Zenon senden
- Befehl erfolgreich an Zenon übertragen: Schicke ACK2 an Control

Diese vier Schritte werden bei Erhalt eines Befehls in die Datei Sicherung.dat geschrieben unter folgender Konvention:

- Name des Befehls vom Control
- ACK1
- Kommando
- ACK2

Die Kommandokette selbst, die die Transportsteuerung an Zenon sendet wird serialisiert in die Datei Kommando.dat geschrieben. In dieser Datei stehen nacheinander alle Befehle, die aus dem Kommando des Controls resultieren.

Kommando.dat wird nach jedem einzelnen Schritt aktualisiert. Das bedeutet, dass ausgeführte Befehle aus der Datei entfernt werden. Sind keine Befehle mehr zu senden wird Kommando.dat gelöscht. Anschließend wird ACK2 an das Control gesendet und auch die Datei Sicherung.dat wird gelöscht.

Wiederherstellungs-Szenario:

Wird die Transportsteuerung gestartet passiert folgendes:

Falls eine Sicherung.dat vorhanden ist wird ein Dialog angezeigt in dem zwischen Initialzustand und Recovery ausgewählt werden kann. (Noch auszuführende Befehle werden dabei in einem Fenster angezeigt). Ist keine Sicherung.dat vorhanden (alle Befehle waren abgearbeitet) wird einfach der Initialzustand angefahren.

Wird die Recovery aktiviert gibt die Sicherung.dat vor, an welcher Stelle weitergemacht wird.

14.3 Konstruktion des Transportsystems

1. Steht ACK1 an 2. Stelle wird dieser gesendet und diese Zeile gelöscht. (in der ersten Zeile bleibt bis zum Ende der Befehl stehen) –; weiter bei 2.
2. Steht Kommando an 2. Stelle wird Kommando.dat geladen und die dort gespeicherte Queue zum abarbeiten übergeben. Ist der letzte Kommando.dat-Befehl hier beendet wird diese Zeile und auch Kommando.dat gelöscht. – weiter bei 3.
3. Steht nun ACK2 in der 2. Zeile von Sicherung.dat wird AKC2 an das Controll gesendet und nun auch die Sicherung.dat gelöscht – Recovery beendet.

Aufbau von Kommando.dat

Diese Datei enthält die Kommandos, welche zum Zeitpunkt des Absturzes als Befehle an die Zenon-SPS zu senden waren. Ein Dreierblock bestehend aus dem Kommando, dem erwarteten ACK1 und ACK2 bilden dabei eine Einheit. Wichtig ist dabei, dass ein Kommando, welches als Message-ID eine „null“ enthält, vor dem Absturz noch nicht gesendet wurde. Dieses Kommando wird im Recovery-Fall gesendet und anschließend gelöscht. Die Antworten ACK1 und ACK2 mit Message-ID „null“ werden von der Zenon-SPS als nächstes erwartet.

14.4 Zenon

Das Prinzip der Visualisierung in unserem Programm beruht auf der Verknüpfung der Visualisierungssymbole mit den eingelesenen Variablen des SPS-Programms. Die Visualisierungssymbole führen, vom Zustand der verknüpften Variable abhängig, eine voreingestellte Aktion aus. Dies könnte z.B. ein Farbwechsel oder eine wechselnde Sichtbarkeit sein.

14.4.1 Variablen

Zum erfassen, steuern und regeln von Prozessen ist einerseits der Austausch von Prozessdaten zwischen einer Steuerung und dem Leitsystem und andererseits die Vorgabe von Sollwerten und Befehlen notwendig. Prozessvariablen repräsentieren den Wert in Steuerung und Visualisierung und bestehen aus einer Summe von projektierbaren Eigenschaften (z.B. Name, Nachkommastellen, Grenzwert, etc.). Die Variablen werden in der zentralen Variablenliste bei jedem Projekt projektiert bzw. parametriert und sind dort von überall (Funktionen, Bilder, Archivierung etc.) her zugänglich. Der Variabtentyp *Formel* verweist auf Mathematikvariablen. Der Wert einer Mathematikvariable kann sowohl mathematisch berechnet als auch mit Hilfe von Logik Operationen (AND, OR, XOR, NOT) bestimmt werden.

Die folgende Tabelle listet alle verwendeten Variablen auf.

Variablenbezeichnung	Kommentar in SPS	DB in SPS	ByteAddr (Offset)	BitNr.	Variabtentyp
Stopper 3 vor	Stopper 3 vor ST3		16	4	Ausgang
Stopper 4 vor	Stopper 4 vor ST4		16	6	Ausgang
Stopper 5 vor	Stopper 5 vor		21	2	Ausgang
Stopper 6 vor	Stopper 6 vor ST6		17	4	Ausgang
Stopper 7 vor	Stopper 7 vor ST7		21	6	Ausgang
Stopper 8 vor	Stopper 8 vor ST8		20	0	Ausgang
Stopper 1 vor	Stopper 1 vor		16	0	Ausgang
Initiator B15	Stopper 15 belegt		24	6	Eingang
Initiator B15.1	Stopper 15 frei		24	7	Eingang
Initiator B16	Stopper 16 belegt		24	2	Eingang
Initiator B16.1	Stopper 16 frei		24	3	Eingang
Weiche 1 geschl. B11.1	Weiche 1 geschl. B11.1		9	0	Eingang
Weiche 1 offen B11	Weiche 1 offen B11		9	1	Eingang

14.4 Zenon

Weiche 2 geschl. B13.1	Weiche 2 ge-schl. B13.1		5	5	Eingang
Weiche 2 offen B13	Weiche 2 offen B13		5	6	Eingang
Stopper 9 vor	Stopper 9 vor ST9		21	4	Ausgang
Stopper S11	Stopper 11 (true=einfahren)		24	5	Ausgang
Stopper S13	Stopper 13 nach Fixierstati-on Lager(true = eingefahren)		24	3	Ausgang
Stopper S15	Stopper S15 Fixierstati-on Roboter(true=einfahren)		24	0	Ausgang
Stopper S16	Stopper 16 Fixierstation Lager (true=einfahren)		24	4	Ausgang
Stopper S17	Stopper S17 Vor Fixier-station Roboter (true=einfahren)		24	1	Ausgang
Stopper 10 vor	Stopper 10 vor ST10		17	2	Ausgang
Stopper 2 vor	Stopper 2 vor		16	2	Ausgang
Initiator B10	Stopper 10 be-legt		8	0	Eingang
Initiator B17	Stopper 17 frei		24	5	Eingang
Initiator B2	Stopper 2 be-legt		4	3	Eingang
Initiator B2.4	Stopper2 frei		4	6	Eingang
Initiator B3	Stopper 3 be-legt		4	7	Eingang
Initiator B3.4	Stopper3 frei		5	1	Eingang
Initiator B4	Stopper 4 be-legt B4		5	2	Eingang
Initiator B4.1	Stopper 4 frei		5	3	Eingang
Initiator B2.1	Palette auf Band 1		4	4	Eingang
Initiator B1	Stopper 1 be-legt		4	0	Eingang
Initiator B1.1	Stopper1 frei		4	1	Eingang
Initiator B1.4	Am Ende von Kurve 2		4	2	Eingang

14.4 Zenon

Initiator B18	Stopper 11 belegt		24	1	Eingang
Initiator B18.1	Am Ende von Kurve1		24	0	Eingang
Initiator B11.2	Am Ende von Kurve1		5	4	Eingang
Initiator B19	Stopper 13 frei		24	4	Eingang
K002_FixierstStellung			46	0	SPS-Merker
K002_Fixierst_aktivieren			44	1	SPS-Merker
K002_FixierstId			45	0	SPS-Merker
Initiator B9	Stopper 9 belegt		9	4	Eingang
Initiator B9.1	Stopper9 frei		9	5	Eingang
Initiator B3.3	Palette auf Band 3		8	7	Eingang
Initiator B5	Stopper 5 belegt		9	2	Eingang
Initiator B5.1	Stopper5 frei		9	3	Eingang
Initiator B7	Stopper 7 belegt		9	7	Eingang
Initiator B14	Kurve Band 3		9	6	Eingang
Initiator B10.1	Stopper10 frei		8	1	Eingang
Initiator B2.3	Palette auf Band 2		5	7	Eingang
Initiator B6	Stopper 6 belegt		8	2	Eingang
Initiator B6.1	Stopper 6 frei		8	3	Eingang
Initiator B8	Stopper 8 belegt		8	6	Eingang
Initiator B8.1	Stopper 8 frei		8	4	Eingang
Band1_Motor			12	2	Ausgang
Band2_Motor			12	1	Ausgang
Band3_Motor			12	0	Ausgang
A002_SchlittenId			50	0	SPS-Merker
A002_Befehl_ausgef			51	1	SPS-Merker
A001_Befehl_verstanden			49	1	SPS-Merker
K003_TS_herunterfahren			48	0	SPS-Merker
F002_FehlerId			54	0	SPS-Merker
F001_Befehl_n_verstanden			52	1	SPS-Merker
F002_Befehl_n_ausgef			53	1	SPS-Merker
K001_Weiche_stellen			40	1	SPS-Merker
K001_Weichenstellung			42	0	SPS-Merker
K001_WeichenId			41	0	SPS-Merker
DB88.SchlittenID		88	14	0	Erw. Datenbaustein

14.4 Zenon

DB86.SchlittenID		86	14	0	Erw. Datenbaustein
DB85.SchlittenID		85	14	0	Erw. Datenbaustein
DB84.SchlittenID		84	14	0	Erw. Datenbaustein
DB83.SchlittenID		83	14	0	Erw. Datenbaustein
DB82.SchlittenID		82	14	0	Erw. Datenbaustein
DB81.SchlittenID		81	14	0	Erw. Datenbaustein
DB80.SchlittenID		80	14	0	Erw. Datenbaustein
DB79.SchlittenID		79	14	0	Erw. Datenbaustein
DB73.SchlittenID		73	14	0	Erw. Datenbaustein
DB72.SchlittenID		72	14	0	Erw. Datenbaustein
DB71.SchlittenID		71	14	0	Erw. Datenbaustein
DB70.SchlittenID		70	14	0	Erw. Datenbaustein
DB69.SchlittenID		69	14	0	Erw. Datenbaustein
DB68.SchlittenID		68	14	0	Erw. Datenbaustein
DB74.SchlittenID		74	14	0	Erw. Datenbaustein
DB99.SchlittenID		99	14	0	Erw. Datenbaustein
DB98.SchlittenID		98	14	0	Erw. Datenbaustein
DB97.SchlittenID		97	14	0	Erw. Datenbaustein
DB96.SchlittenID		96	14	0	Erw. Datenbaustein
DB94.SchlittenID		94	14	0	Erw. Datenbaustein
DB93.SchlittenID		93	14	0	Erw. Datenbaustein
DB92.SchlittenID		92	14	0	Erw. Datenbaustein

14.4 Zenon

DB100.SchlittenID		100	14	0	Erw. Datenbaustein
DB91.SchlittenID		91	14	0	Erw. Datenbaustein
DB90.SchlittenID		90	14	0	Erw. Datenbaustein
DB89.SchlittenID		89	14	0	Erw. Datenbaustein
Schlitten auf Band1 ausschleusen					Formel
Schlitten auf Band3 ausschleusen					Formel
Schlitten auf Band2 ausschleusen					Formel
Schlitten an Fixierstation fixieren					Formel
Schlitten an Fixierstation loslassen					Formel
Fix_1_an	wenn ausgang auf true/1 dann fährt fix an lager aus		24	6	Ausgang
Fix_2_an	wenn ausgang auf true/1 dann fährt fix an roboter aus		24	2	Ausgang
Fix_0_an	wenn ausgang auf true/1 dann fährt fix an e/a aus		17	6	Ausgang
Befehl ausgeführt					Formel
Warte auf Acknowledge 2					Formel
Warte auf Acknowledge 1					Formel
Warte auf Befehl					Formel
K001_SchlittenId		36	0		SPS-Merker
K002_SchlittenId		38	0		SPS-Merker

14.4 Zenon

14.4.2 Funktionen

Die Eingriffe der Benutzer in das Zenon-Leitsystem werden über benutzerdefinierte Projektfunktionen realisiert. Alle in einem Projekt verwendeten Funktionen basieren auf den vorhandenen Systemfunktionen (siehe Abbildung 14.3).

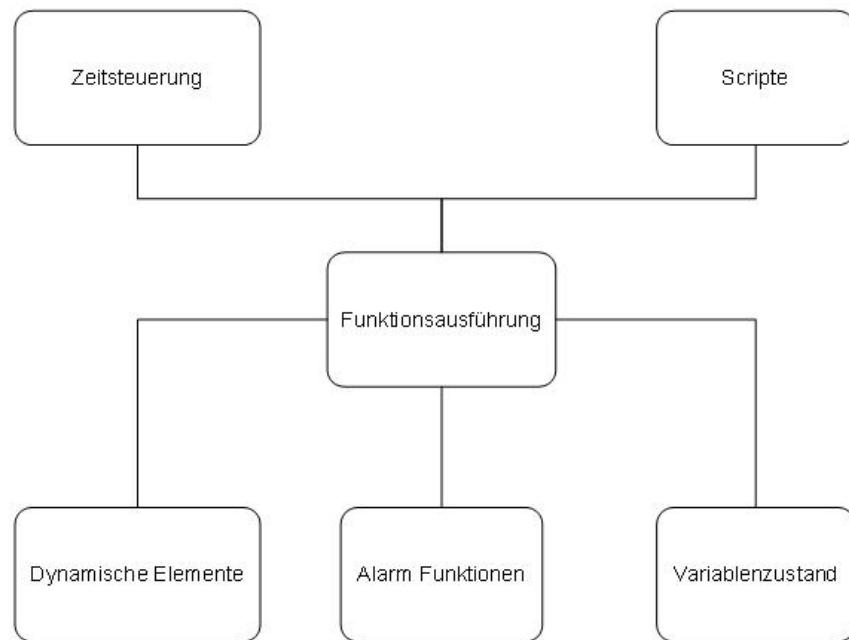


Abbildung 14.3: Funktionsausführung

Die Liste der definierten Funktionen des aktiven Projekts wird nach der Anwahl der Verzweigung *Funktionen* in der Baumsicht des Projektmanagers in der Detailansicht angezeigt (siehe Abbildung 14.4).

14.4 Zenon

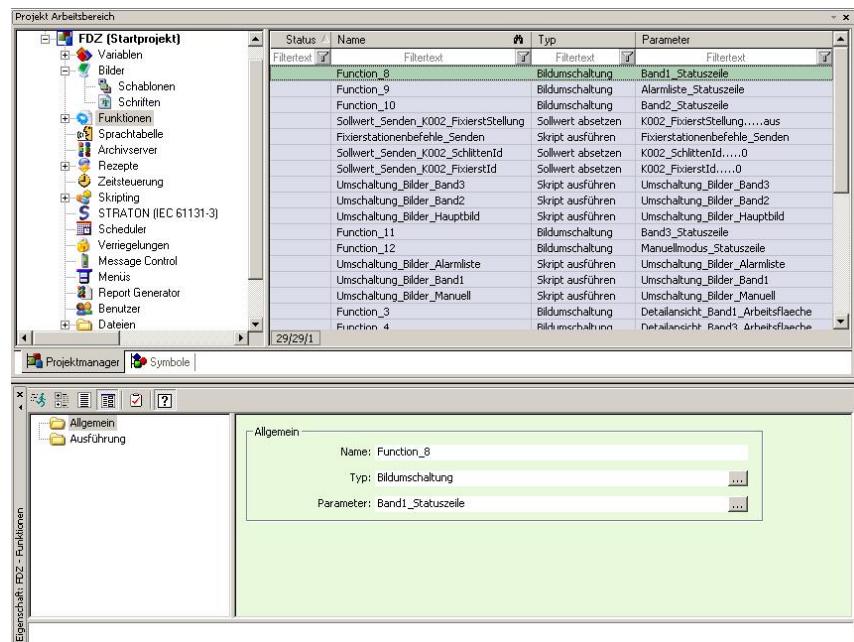


Abbildung 14.4: Darstellung der Funktionsliste in Zenon

Die von uns angelegten Funktionen und die von ihnen ausgeführten Aktionen sind in der folgenden Tabelle aufgelistet.

Funktionsbezeichnung	Setzt Variable	Führt Skript aus	Wechselt auf Bild
Sollwert_Senden_K002_-Fixierst_aktivieren	K002_Fixierst_aktivieren		
Weichenbefehle_Senden		Weichenbefehle_Senden	
Sollwert_Senden_K001_-Weichenstellung	K001_Weichenstellung		
Sollwert_Senden_K001_-WeichenId	K001_WeichenId		
Sollwert_Senden_K001_-Weiche_stellen	K001_Weiche_stellen		
Sollwert_Senden_K001_-SchlittenID	K001_SchlittenId		
Function_7			Alarmliste
Function_6			Manuellmodus_Arbeitsflaeche
Function_5			Detailansicht_Band2_Arbeitsflaeche
Function_4			Detailansicht_Band3_Arbeitsflaeche
Function_3			Detailansicht_Band1_Arbeitsflaeche

14.4 Zenon

Function_2			Bild_Menuleiste
Function_1			Hauptbild_Statuszeile
Function_0			Hauptbild_Arbeitsflaeche
Umschaltung_Bilder_Manuell		Umschaltung_Bilder_Manuell	
Umschaltung_Bilder_Hauptbild		Umschaltung_Bilder_Hauptbild	
Umschaltung_Bilder_Band3		Umschaltung_Bilder_Band3	
Umschaltung_Bilder_Band2		Umschaltung_Bilder_Band2	
Umschaltung_Bilder_Alarmliste		Umschaltung_Bilder_Alarmliste	
Umschaltung_Bilder_Band1		Umschaltung_Bilder_Band1	
Function_12			Manuellmodus_Statuszeile
Function_11			Band3_Statuszeile
Function_10			Band2_Statuszeile
Function_9			Alarmliste_Statuszeile
Function_8			Band1_Statuszeile
Fixierstationenbefehle_Senden		Fixierstationenbefehle_Senden	
Sollwert_Senden_K002_-SchlittenId	K002_SchlittenId		
Sollwert_Senden_K002_-FixierstStellung	K002_FixierstStellung		
Sollwert_Senden_K002_-FixierstId	K002_FixierstId		

14.4 Zenon

14.4.3 Skripte

Sollen mehrere benutzerdefinierte Funktionen zu einer Abfolge verbunden werden, müssen sie zu einem Skript zusammengefasst werden. Hierbei können alle Funktionen verwendet werden. Im System sind bereits mehrere Skripte definiert, die nach Bedarf abgearbeitet werden können. Wir haben folgendes, vom System bereit vorgegebene Skript verwendet:

- **AUTOSTART**: Das Skript wird beim Starten der Runtime automatisch als Startbild ausgeführt. In unserem Fall führt dieses Skript die Funktionen aus, die das Hauptbild darstellen.

Alle anderen Skripte wurden von uns selbst erstellt. Abbildung 14.5 zeigt die Auflistung der verwendeten Skripte in Zenon.

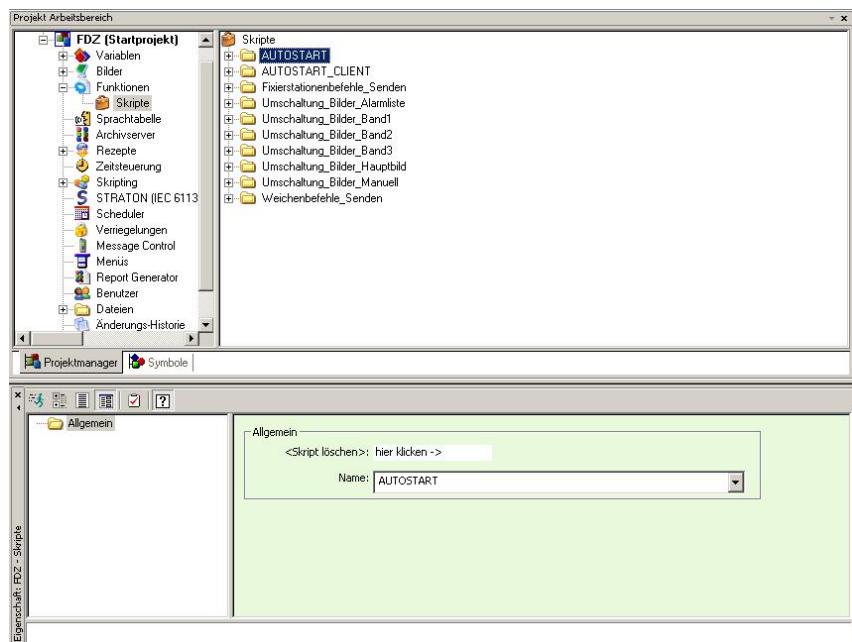


Abbildung 14.5: Darstellung der verwendeten Skripte in Zenon

In der folgenden Tabelle sind alle verwendeten Skripte und die darin zusammengefassten Funktionen aufgeführt.

Skript	Führt folgende Funktionen aus
Fixerstationenbefehle_Senden	Sollwert_Senden_K002_FixerstStellung; Sollwert_Senden_K002_FixerstId; Sollwert_Senden_K002_Fixerst_aktivieren; Sollwert_Senden_K002_SchlittenId;
Umschaltung_Bilder_Manuell	Function_12;Function_6;
AUTOSTART_CLIENT	Function_3;Function_4;Function_5; Function_6;Function_7;

14.4 Zenon

Weichenbefehle_Senden	Sollwert_Senden_K001.SchlittenID; Sollwert_Senden_K001_Weiche_stellen; Sollwert_Senden_K001_WeichenId; Sollwert_Senden_K001_Weichenstellung;
Umschaltung_Bilder_Band1	Function_8;Function_3;
Umschaltung_Bilder_Band2	Function_10;Function_5;
Umschaltung_Bilder_Band3	Function_11;Function_4;
Umschaltung_Bilder_Hauptbild	Function_0;Function_1;
Umschaltung_Bilder_Alarmliste	Function_7;Function_9;
AUTOSTART	Function_0;Function_1;Function_2;

14.4.4 Bilder und Schablonen

Hauptbestandteil eines Projektes sind die Bilder, die dem Bediener einen möglichst umfassenden Informationsstand über die zu bedienenden Anlagen geben sollen. Mögliche Bilder sind z.B. Prozess-, System- und Bedienbilder.

Während der Projektierungsphase werden diese Bilder erstellt und für den späteren Online-Betrieb vorbereitet. Das Leitsystem bietet die Möglichkeit, mehrere Bilder gleichzeitig in verschiedenen Fenstern und auf mehreren Monitoren zu beobachten und zu bedienen. Für die einzelnen Bilder erfolgt die Definition von Größe, Position und spezifischen Eigenschaften (Bildscript, Hintergrundfarbe) mit Hilfe des Editors. Mittels zentral in der Datenbank verwalteten Fensterdefinitionen (Schablonen) können die Aufschaltparameter für die Bilder vordefiniert werden.

In Abbildung 14.6 sind alle erstellten Bilder in Zenon aufgelistet.

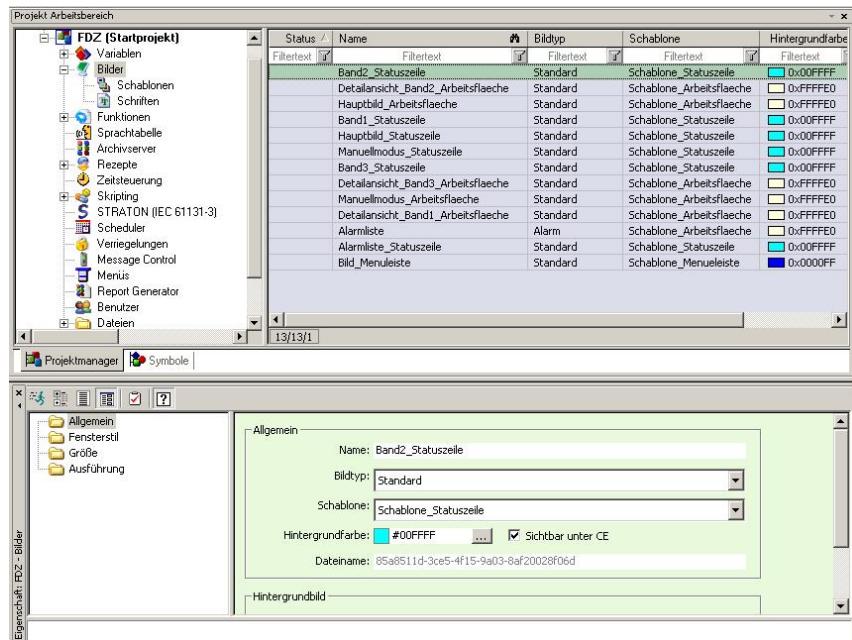


Abbildung 14.6: Darstellung der Bilderliste in Zenon

14.4 Zenon

Die Gestaltung und der Aufbau der Schablonen und Prozessbilder kann frei durchgeführt werden. Projektspezifische Änderungen oder Erweiterungen können ebenfalss jederzeit vorgenommen werden.

In den Eigenschaften von Bildern stehen für die Prozessbilder unterschiedliche Bildtypen zur Verfügung. In diesem Projekt wurden nur zwei Bildtypen gewählt, *Standard* zur Prozessdarstellung und *Alarm* zur Darstellung von Alarmlisten.

Diese Einstellung wurde in den Eigenschaften im Ordner *Allgemein*, Feld *Allgemein* und Zeile *Bildtyp* vorgenommen (siehe Abbildung 14.7).



Abbildung 14.7: Bildtypauswahl

Die Schablonentechnik stellt die Grundlage für die Fensterdefinition dar, da jedes Prozessbild und jede Prozesstabellen innerhalb einer Schablone aufgeschaltet wird. In den Schablonen werden die generellen Fenstereigenschaften (Position, Größe, Styling und die Manipulationsmöglichkeiten zur Laufzeit) definiert. Die Schablonentechnik bietet folgende Vorteile:

- Übersicht über alle in einem Projekt definierten Fenster
- Funktionen können speziellen Schablonen zugewiesen werden (Bild quittieren, Bild abwählen etc)
- Eine Änderung einer Schablone ändert auch alle Bilder, die auf dieser Schablone basieren

Interaktives Verschieben oder Vergrößern kann im Sinne einer sicheren Prozessführung unterbunden werden. Die Prozessbilder und -tabellen erscheinen somit immer in der gewohnten Größe und an der vordefinierten Stelle am Monitor. Die Bilder innerhalb einer Schablone können im Onlinebetrieb beliebig ausgetauscht werden. Das Erstellen der Schablonen erfolgt mit dem Editor im Projektmanager unter der Verzweigung *Bilder* und dem Eintrag *Schablonen*. Das Kontextmenü bietet den Eintrag *Schablone neu*. Daraufhin wird im Eigenschaften-Fenster die neue Schablone definiert (siehe Abbildung 14.8).

14.4 Zenon



Abbildung 14.8: Neue Schablone erstellen

Für jede Schablone sind mehrere Eigenschaften definierbar, die wichtigsten sind *Name* und *Größe der Schablone am Monitor*. Auf diese Weise wurden drei Schablonen für das Projekt erstellt (siehe Abbildung 14.9).

14.4 Zenon

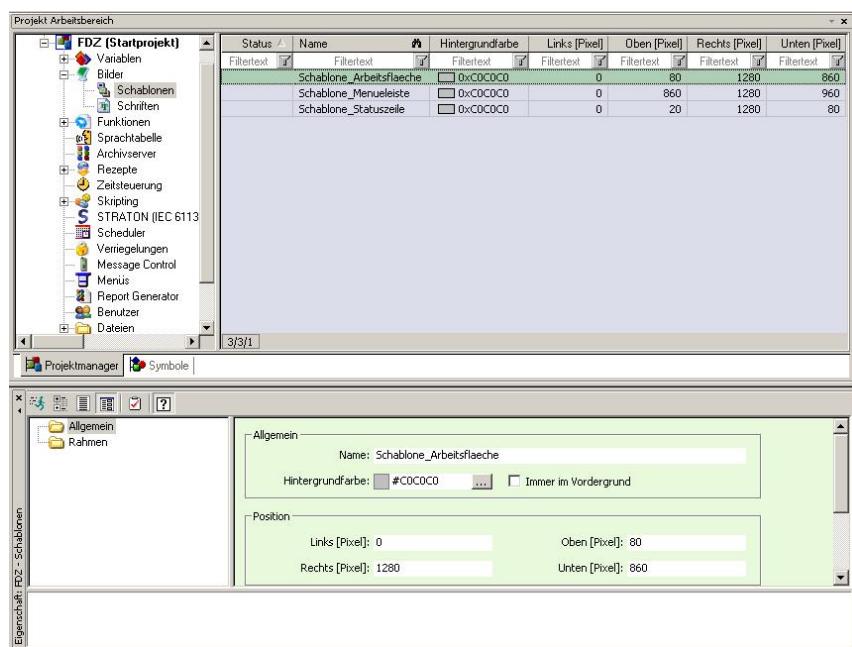


Abbildung 14.9: Erstellte Schablonen

So wurde z.B. die Größe der Schablone für die Statuszeile am oberen Bildschirmrand wurde auf 60x1280 eingestellt (siehe Abbildung 14.10).

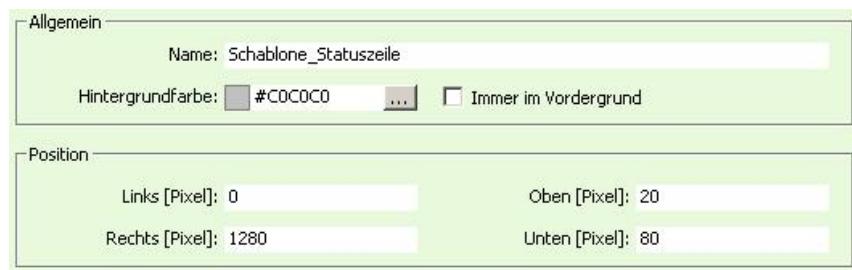


Abbildung 14.10: Schablone für die Statuszeile

Die Schablone für die Arbeitsfläche in der Mitte des Monitors hat die Größe 780x1280 und die Schablone für die Menüzeile am unteren Bildschirmrand 100x1280.

14.4 Zenon

14.4.5 Alarmauswertung

Zur Visualisierung der möglichen Fehler die auftreten können haben wir das von Zenon bereits vorgefertigte Kontrollelement *Alarmliste* verwendet. Um die auftretenden Fehlermeldungen auszuwerten haben wir eine Reaktionsmatrix mit Namen *Alarm* erstellt und diese als Grenzwerte der Variable F002_FehlerId eingesetzt. Mit Hilfe von Reaktionsmatrizen ist eine umfangreiche Verarbeitung von Stati möglich. Hier können einzelne Stati ausgewertet werden und Alarme auslösen.

Um zu erreichen, dass die in der Reaktionsmatrix eingebenen Meldungen in der Alarmliste erscheinen haben wir, wie in Abbildung 14.11 sichtbar in den Feldern *als Alarm*, *quittier-* und *löschenpflichtig* ein Haken gesetzt. Der Haken im Feld *in CEL* bewirkt, dass die aktuelle Fehlermeldung immer am oberen Bildschirmrand rot angezeigt wird, gleichgültig in welchem Bild man sich befindet.

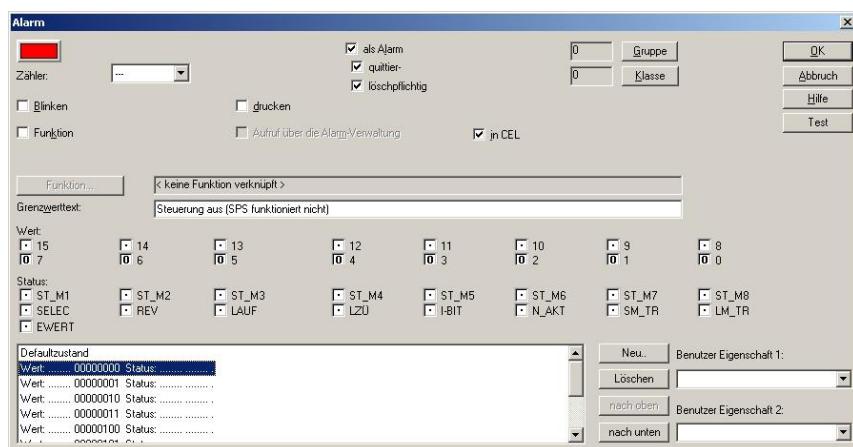


Abbildung 14.11: Alarm-Reaktionsmatrix

Die folgende Tabelle zeigt die möglichen FehlerIds und deren Meldungen.

FehlerID	Fehlermeldung
0	Steuerung aus (SPS funktioniert nicht)
1	kein Druck vorhanden (Weichenstellung funktioniert nicht)
2	Sicherungsfall
3	Antrieb Band 1 Störung
4	Antrieb Band 2 Störung
5	Antrieb Band 3 Störung
6	Not-Aus betätigt
7	Timeout
8	kein freier Schlitten d.h. Benutzer muss neuen Schlitten einsetzen
I-BIT	Steuerung aus (SPS funktioniert nicht)

FehlerId 9-255 ist der Defaultzustand der Reaktionsmatrix, d.h. es wird keine Alarmmeldung ausgegeben. Das I-Bit wird nur dann gesetzt wenn keinerlei Verbindung mehr zur SPS besteht

14.4 Zenon

und somit keine FehlerId mehr ausgelesen werden kann.

14.4.6 Visualisierungssymbole

Alle angelegten Symbole die mit einer Variablen verknüpft werden, könnten während des Betriebes vom Benutzer verändert werden. Da die GUI aber nur zur Visualisierung dienen soll wurde diese Möglichkeit in allen Bildern gesperrt. Nur im *Manuellen Modus* besteht weiterhin die Möglichkeit die Werte der Variablen zu verändern.

14.4.6.1 Darstellung SchlittenId und -position

Während des Betriebes des Bandes werden die aktuellen Positionen der sich auf den Bändern befindenden Schlitten mit der dazugehörigen Id dargestellt (siehe Abbildung 14.12).

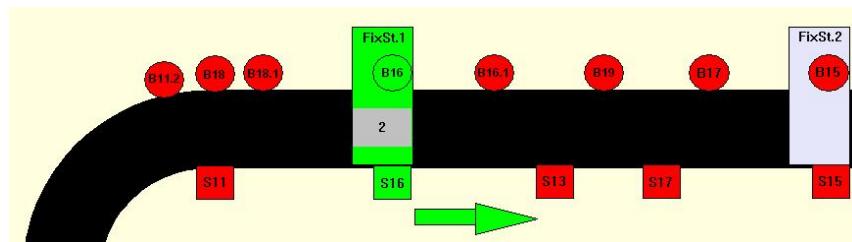


Abbildung 14.12: Darstellung Schlittenposition mit Id

Vorgehensweise:

Um das Symbol für den Schlitten zu erstellen, wurde ein *Dym.element-Zahlenwert* in den Bildern *Hauptansicht*, *Detailansicht Band 1,2 und 3* angelegt. Durch die Verknüpfung mit der dazugehörigen *SchlittenId*-Variable kann der aktuelle Istwert der Variable angezeigt werden. Um die Sichtbarkeit einer Variable einzustellen wurde in den Eigenschaften des Zahlenwertes im Ordner *Sichtbarkeit/Blinken* diese Variable eingetragen und der Haken im Feld von *Grenzwert übernehmen* entfernt. Für die Grenzen wurde der Bereich von 0 bis 32767 eingestellt, da alle *SchlittenIds* die im Prozessablauf nicht aktiv sind von der SPS mit -1 beschrieben werden und das Feld somit nicht mehr sichtbar sein soll (siehe Abbildung 14.13).

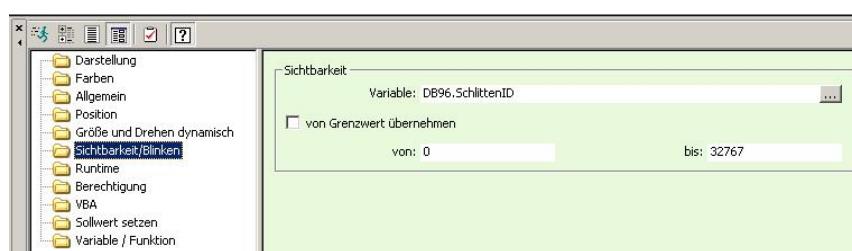


Abbildung 14.13: Schlitteneigenschaften

14.4 Zenon

14.4.6.2 Darstellung Sensoren, Stopper, Bänder und Fixierstationen

Die verwendeten Symbole für Sensoren, Stopper, Bänder und Fixierstationen dienen rein der Anzeige des aktuellen Zustandes der verknüpften Variable. Durch das Erreichen des voreingestellten Grenzwertes der Variable wird ein Farbwechsel der Symbole ausgeführt (siehe Abbildung 14.14).

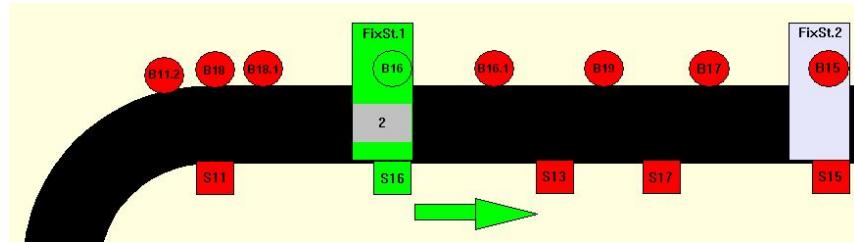


Abbildung 14.14: Darstellung Sensoren, Stopper, Bänder und Fixierstationen

Fallbeispiel mit Sensor B16:

Um das Symbol für den Sensor zu erstellen, wurde ein *Vekt.element-Kreis* in den Bildern *Hauptansicht*, *Detailansicht Band 1,2 und 3* angelegt und unter den Eigenschaften des angelegten Kreises im Ordner *Farben*, im Feld *Farben Dynamisch* mit der Variable *Initiator B16* in der Zeile *Hintergrund-/Füllfarbe* verknüpft (siehe Abbildung 14.15).

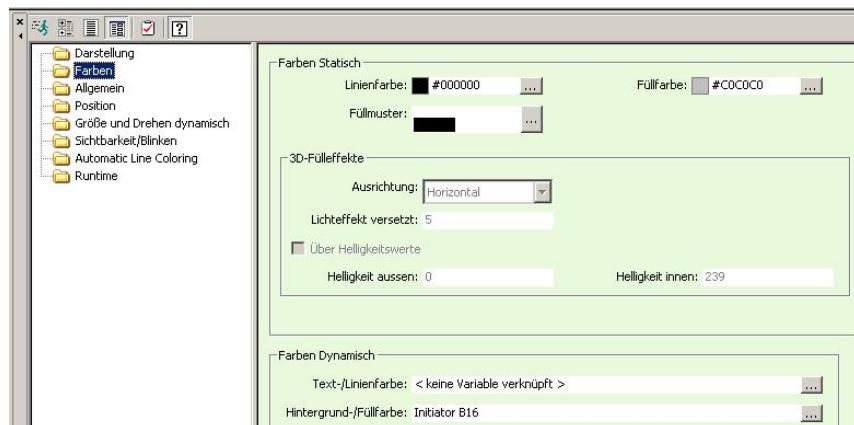


Abbildung 14.15: Sensoreigenschaften

Um den gewünschten Farbwechsel einzustellen, wurde in der Variable *Initiator B16* in den Eigenschaften im Ordner *Grenzwerte* im Feld *Grenzwert1* der Punkt *Grenzwert aktiv* ausgewählt (siehe Abbildung 14.16).

14.4 Zenon

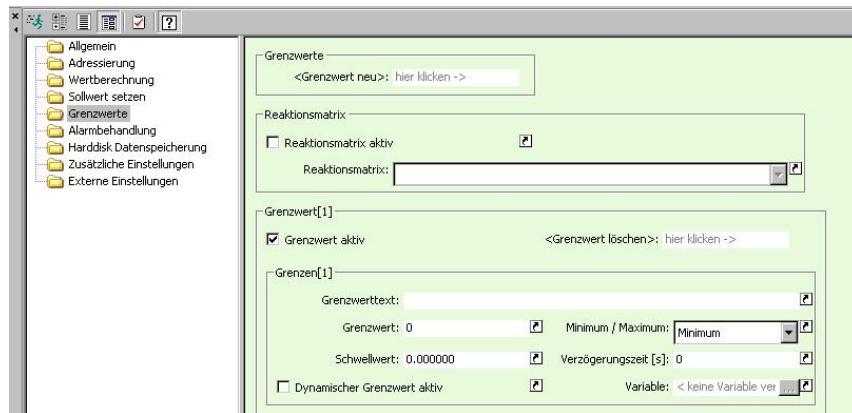


Abbildung 14.16: Grenzwert Einstellung

Daraufhin wurde in der Zeile *Grenzwert* der Wert 0 eingegeben und im Feld *Zusatzattribute1*, in der Zeile *Farbe* die Farbe *Rot* ausgewählt (siehe Abbildung 14.17).

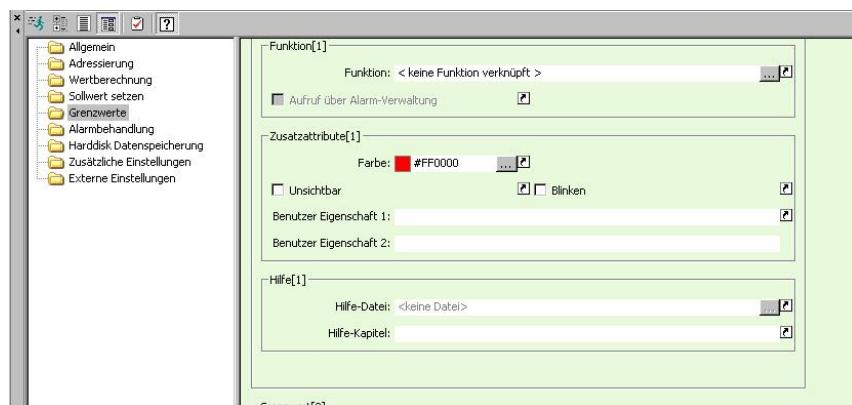


Abbildung 14.17: Farbeinstellung für Grenzwert

Im Feld *Grenzwert2* wurde der Punkt *Grenzwert aktiv* ausgewählt, in der Zeile *Grenzwert* der Wert 1 eingegeben und im Feld *Zusatzattribute2* in der Zeile *Farbe* die Farbe *Grün* ausgewählt.

Nach diesen vorgenommenen Einstellungen nimmt das angelegte Symbol je nach Zustand der Variable die entsprechende Farbe an.

14.4.6.3 Darstellung Status und Befehle

Da im Feld *aktueller Befehl* und *Status* alle möglichen Befehle und Stati an der selben Position angezeigt werden müssen ist es hier erforderlich die Textfelder nur unter bestimmten Bedingungen anzuzeigen (siehe Abbildung 14.18).



Abbildung 14.18: Darstellung aktueller Befehl und Status

Da die Einstellung einer Sichtbarkeit nur mit einer einzigen Variable verknüpfbar ist und somit keine Auswahlbedingungen möglich sind, wurden hierfür eine Mathematikvariablen verwendet. Mit Hilfe der Mathematikvariablen, die selbst erstellt werden müssen, konnten mittels Logik Operationen (AND, OR, XOR, NOT) Auswahlbedingungen definiert werden. Treffen die Bedingungen zu, nimmt die Mathematikvariable den Wert 1 an.

So sollte z.B. der Text *K001 Schlitten auf Band1 ausschleussen* (siehe Abbildung 14.19) nur angezeigt werden, wenn die Variable *K001_Weiche_stellen* den Wert 1 und die Variablen *K001_WeichenId* und *K001_Weichenstellung* den Wert 0 angenommen haben (siehe Abbildung 14.20). Also wurde die Sichtbarkeit des Textfeldes mit der Mathematikvariable verknüpft die den Wert 1 annimmt wenn die Bedingungen zutreffen.



Abbildung 14.19: Anzeige aktueller Befehl



Abbildung 14.20: Variablenwerte zum aktuellen Befehl

Analog dazu wurde für alle Befehlsanzeigen und Statusanzeigen eine Mathematikvariable angelegt, die bei den entsprechenden Bedingungen den Wert 1 annimmt und mit der Sichtbarkeit des Textfeldes verknüpft.

14.4.6.4 Darstellung Variablen-Istwerte

In der Ansicht *Manueller Modus* werden die Istwerte der wichtigsten Variablen dargestellt und können bei Bedarf vom Benutzer verändert werden. Als Symbole für die Darstellung der Variablen, wurden mehrere *Dym.element-Zahlenwert* im Bild *Manueller Modus* angelegt und mit der

14.5 SPS

dazugehörigen Variable verknüpft, um den aktuellen Istwert der Variable anzuzeigen (siehe Abbildung 14.21).

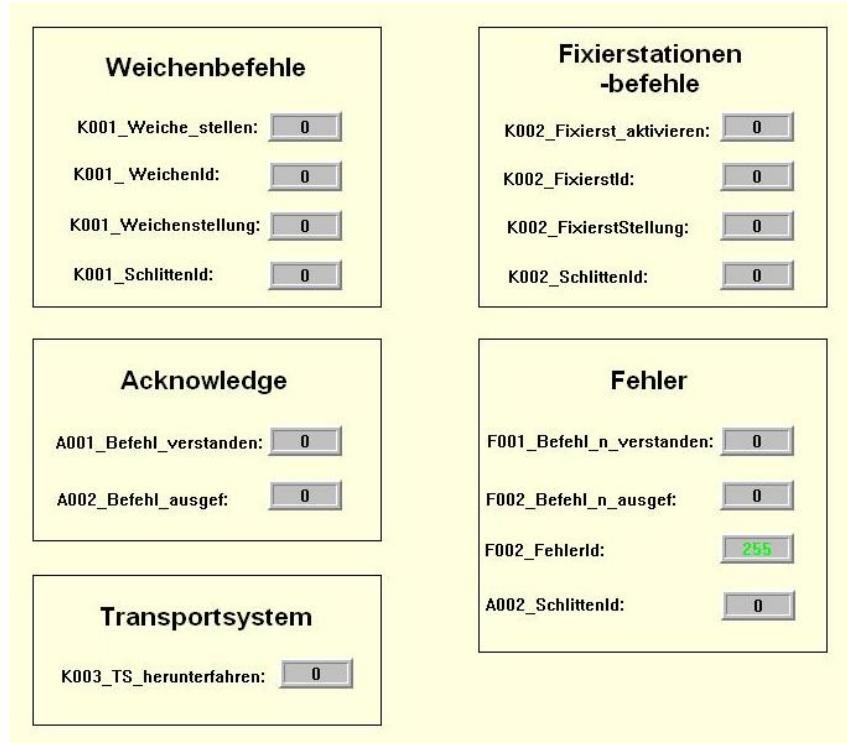


Abbildung 14.21: Visualisierung der Variablen-Istwerte

14.5 SPS

Nach der ordnungsgemäßen Inbetriebnahme befindet sich die SPS im wartenden Zustand. Da die SPS selbst nicht „denken“ muss, sondern nur Befehle von oben entgegennimmt, wird intern laufend über die Kommunikationsvariablen (Merker) gepollt. Diese werden von zenOn geschrieben und anschließend wieder ausgelesen. Wenn ein gültiger Befehl empfangen wurde, wechselt die SPS in den arbeitenden Zustand und arbeitet den Befehl ab. Sobald der Auftrag fertig ist, wird die entsprechende Kommunikationsvariable zurückgesetzt. Daran erkennt zenOn das der Befehl erfolgreich abgearbeitet wurde.

Vereinfacht kann der Ablauf wie folgt dargestellt werden:

- Warten auf Befehl von zenOn (Polling)
- Wenn Befehl erhalten, dann in arbeitenden Zustand wechseln
- Ausführung des Befehls
- Nach Beendigung entsprechende Kommunikationsvariable zurücksetzen
- Im Fehlerfall tritt eine Sonderbehandlung ein

14.5 SPS

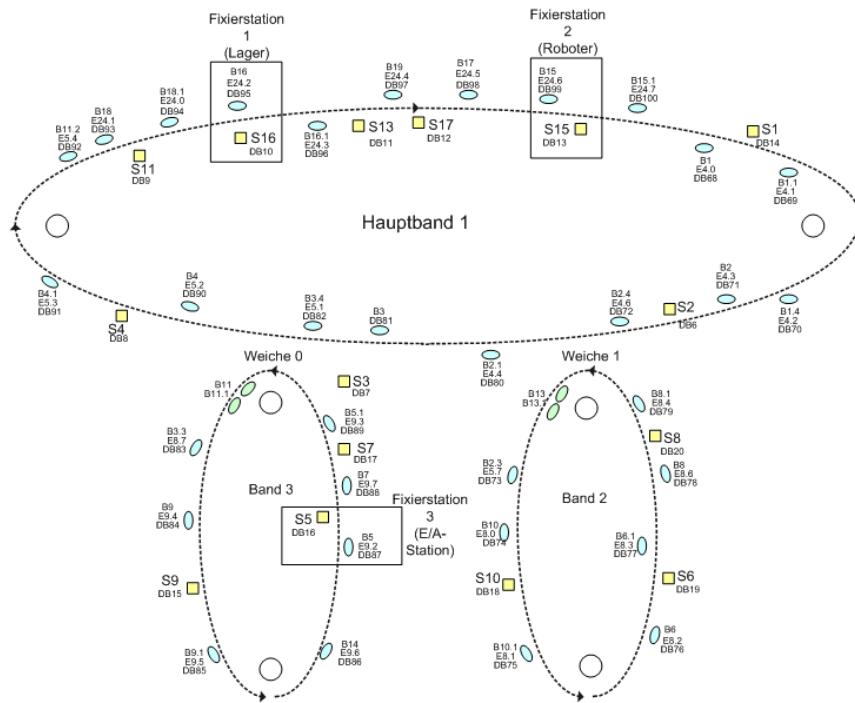


Abbildung 14.22: Schema des Transportsystems mit Sensoren(inklusive zugehörigen DBs) und Stopfern

14.5.1 Die Bänder, FB5

Das Transportband besteht aus drei Bändern: Dem großen Hauptband(Band 1) und zwei kleineren Bändern: Nebenband (Band 2) und E/A-Band (Band 3). Die Bänder sind miteinander verbunden. Die Bewegungsrichtung ist in der Abbildung eingezeichnet, sie kann nicht verändert werden. Nebenband 2 dient als 'Abstellplatz' und als Einsetzort für leere Schlitten. Die ID der Schlitten die dieses Band befahren werden bei Sensor B10 gelöscht und bei Stopper 8 angehalten. Wird ein leerer Schlitten angefordert verlässt ein Schlitten das Nebenband 2 und eine neue ID wird bei Sensor B8.1 generiert. Nebenband 3 führt zur EA-Station. Schlitten die auf diesem Band fahren, bleiben solange auf diesem Nebenband bis ein Befehl zum runterleiten kommt.

14.5.2 Die Nummerierung, FB101-103

Funktionsweise Nummerierung

Die Nummerierung erfordert ein bisschen Querdenken! Man darf es sich nämlich nicht so vorstellen, dass ein Schlitten mit einer bestimmten Nummer initialisiert wird, und damit auf dem Transportband fährt, sondern die Nummer von Sensor zu Sensor übergeben wird, wenn der Schlitten darüber fährt. Das heißt, die Sensoren warten mit den Schlittennummern, bis diese dort ankommen. Jedem Sensor ist ein eindeutiger Datenbaustein zugewiesen (siehe Abb. ??). In diesen Datenbausteinen werden bekanntlich die verschiedenen Variablen abgespeichert. Die

14.5 SPS

für die Nummerierung verwendeten Funktionsbausteine sind FB101 (Initialisierung der Schlitten), FB102 (ID-Verwaltung) und FB103(ID-Weitergabe).

1. Nummerierung allgemein: Für die Beschreibung der Nummerierung allgemein sind zunächst folgende Variablen ausreichend: INITIATOREINGANG, NACHFOLGER und SchlittenID In den folgenden Beispielen wird Nummerierung anhand Sensor B1 (zugehöriger Datenbaustein: DB 68) im Speziellen erklärt

- Beispiel 1:

Ausgangssituation:

Schlitten kommt an Sensor erwartet keinen Schlitten (DB68.SchlittenID = -1)

Ablauf:

Ankommenden Schlitten erkennen (FB101) Der ankommende Schlitten löst am INITIATOREINGANG (hier E4.0) eine steigende Flanke aus. Anschließend wird die kleinste freie ID aus dem Nummernpool ausgewählt (über FC110) und dem Schlitten 'zugewiesen', d.h. im Datenbaustein gespeichert(DB68). Die SchlittenID's werden im durch FB102 verwaltet.

Nummernweitergabe (FB103) Wenn der Schlitten den Sensor wieder verlässt wird eine fallende Flanke ausgelöst und die ID an den nächsten Sensor (NACHFOLGER, hier Sensor B1.1, DB69) weitergeschickt. Die SchlittenID wird außerdem im Vorgänger-DB (hier DB68) wieder gelöscht (Rücksetzen auf -1).

- Beispiel 2:

Ausgangssituation:

Schlitten kommt an Sensor erwartet einen Schlitten (DB68.SchlittenID größer -1)

Ablauf:

Analog Beispiel 1, nur wird in diesem Fall keine neue SchlittenID vergeben, da das System (FB101) einen bereits bestehenden Schlitten erkennt.

2. Nummerierung Sonderfälle:

- Nummerierung an Weichen

Für die Nummerierung an Weichen wird zunächst die SchlittenID an 2 nachfolgende Sensoren (Hauptband + Nebenband 1 oder 2) übergeben. Sobald der Schlitten an einem der beiden Nachfolger angekommen ist, wird die ID im zweiten Nachfolger gelöscht. Deshalb wurden zusätzlich die Variablen NACHFOLGER2, Verzweigung, Verzweigungloeschen und SENSOR eingefügt.

Bsp.: Schlitten fährt von Hauptband auf Nebenband 1

Ausgangssituation:

Schlitten verlässt Sensor B3.4

Ablauf:

14.5 SPS

Die SchlittenID wird von Sensor B3.4, DB82 an die Sensoren B4, DB90, NACHFOLGER und B3.3, DB83, NACHFOLGER2 übergeben. Sobald der Schlitten an Sensor B3.3 angekommen ist, wird die SchlittenID von B4 in DB90 gelöscht. Anschließender Programmablauf (Nummernweitergabe) wieder analog in Nummerierung allgemein ??.

Die Verzweigungen von Nebenband 1 aufs Hauptband, sowie an Nebenband 2 funktionieren natürlich nach demselben Prinzip.

- Schlitten auf Nebenband 2 ausschleusen Wird ein Schlitten auf das Nebenband 2 ausgeschleust, wird anschließend automatisch bei Sensor B10 (DB74) die SchlittenID gelöscht und im Nummernpool wieder freigegeben. D.h. die SchlittenID wird auf -1 zurückgesetzt. Der leere Schlitten wartet dann an den Stopfern S8, S6 oder S10, je nachdem wie viele Schlitten sich schon auf dem Nebenband befinden, auf eine Neuanforderung durch die Bandsteuerung. Für den Programmablauf wurden dazu die Variablen: KeinNachfolger und LoeschSchlitten eingeführt.

Ausgangssituation:

Schlitten befindet sich auf Nebenband 2, vor Sensor B10

Ablauf:

Sobald der Schlitten Sensor B10 überfährt, wird mit Hilfe der Funktion FC111 die SchlittenID gelöscht und der Schlitten fährt als 'leerer Schlitten' weiter und wartet am vordersten freien Stopper auf dem Nebenband 2 bis er wieder durch die Bandsteuerung eingesetzt wird.

Theoretisch ist ein Einsetzen von Schlitten überall möglich, solange sich kein Schlitten in einem Bereich zwischen zwei Sensoren befindet ??.

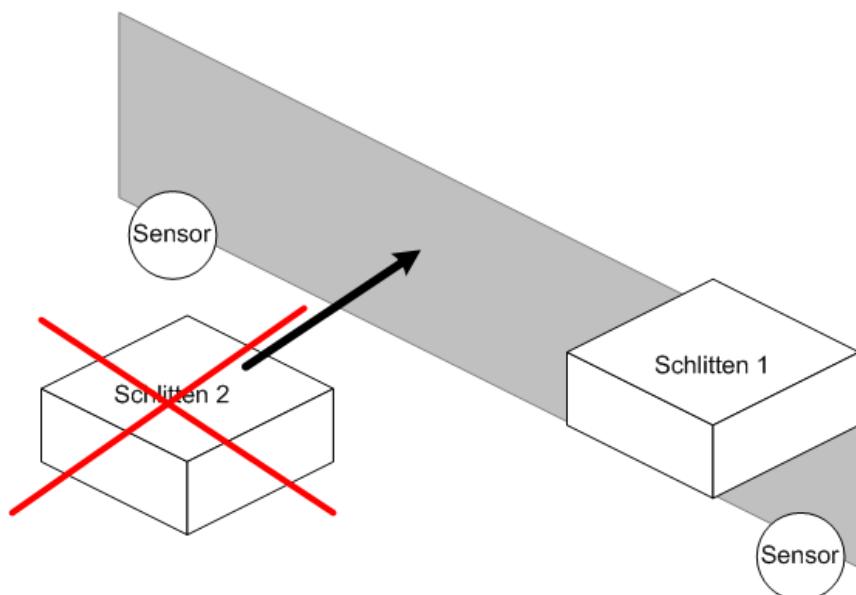


Abbildung 14.23: Falsches Einsetzen von Schlitten

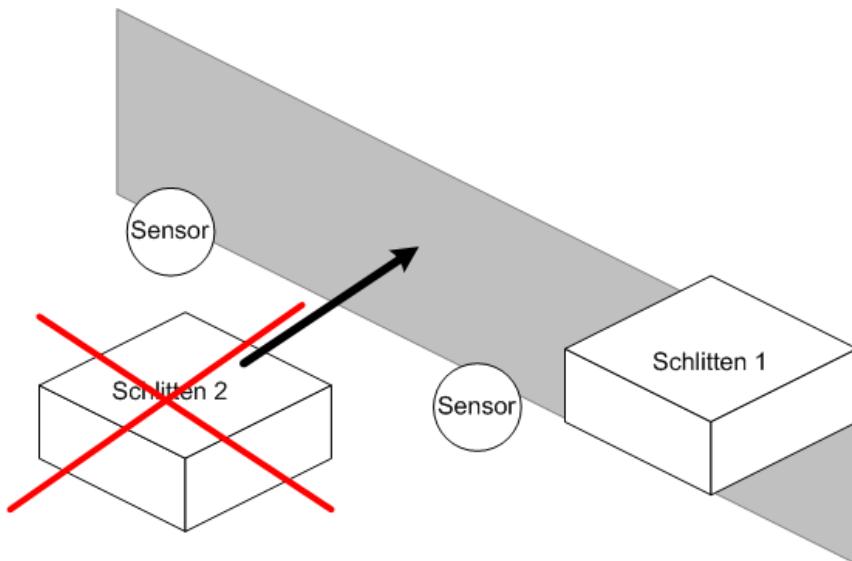


Abbildung 14.24: Falsches Einsetzen von Schlitten

14.5.3 Timeout, OB35, DB35, FB101

Um zu verhindern, dass eingeklemmte Schlitten vom Benutzer unbemerkt bleiben, wurde eine Timeout-Steuerung implementiert, die nach 10 Sekunden eine Fehlermeldung auslöst und das Band stoppt. Besonders gefährdete Stellen am Transportband sind dabei die Weiche, Fixierstationen und Kurven. Ist zwischen 2 Sensoren ein Stopper ausgefahren, wird dies natürlich für den Timeout berücksichtigt!

- Der Timeout wurde wie folgt programmiert.

Die SPS besitzt einen OB35 der alle 100us ein Interrupt auslöst. Zu diesem OB35 gehört der Instanz-Datenbaustein DB35. Im OB35 wurde eine Variable 'Zeit' angelegt, diese wird bei jedem Interrupt um 1 erhöht und ist für den Timeout zuständig.

Die Variable wird initialisiert und läuft endlos von 0 bis 30000.

- Funktionweise des Timeouts:

Wenn ein Schlitten über einen Sensor fährt wird von diesem Sensor bis zum nächsten Sensor der sich auf dem Band befindet (Bandabschnitt) der Schlitten überwacht. Dabei wird geprüft ob der Schlitten innerhalb von 10sec den Bandabschnitt wieder verlässt.

Die Überwachung wurde wie folgt realisiert: Wenn der Schlitten über einen Sensor fährt wird der aktuelle Wert aus der Variable 'Zeit' im DB35 in die Variable 'Startzeit' gespeichert. Nun wird bei jedem Durchlauf die Startzeit mit der aktuellen Zeit verglichen, wird der Zeitunterschied größer als 10sec (=100 Interrupts) wird ein Fehler (FehlerID 7) ausgelöst und das Band bleibt stehen.

Die Timeoutzeit kann im FB101 eingestellt werden siehe Code.

Code: IF (ABS (DB35.Zeit - Startzeit)) größer 100 THEN

14.5 SPS

- Spezialfall:

Befindet sich in einem Bandabschnitt ein Stopper so wird dieser in der Timeout Steuerung berücksichtigt. Da der Schlitten an dem Stopper aufgehalten werden kann und damit die 10sec überschritten würden. Dieser Fall wurde berücksichtigt und wurde wie folgt gelöst.

Sobald ein Stopper ausgefahren ist wird dies erkannt und solange die Startzeit mit der aktuellen Zeit überschrieben, somit wird verhindert das man einen ungewollten Timeoutfall bekommt.

CODE:

```
IF Stopper = true THEN // wenn ein Stopper ausgefahren ist, dann soll die Startzeit zyklisch  
überschrieben werden. Startzeit := DB35.Zeit; // dadurch wird verhindert, dass der Timeout  
an Stopfern ausgelöst werden kann. END_IF;
```

Für genauere Information zum Timeout siehe Programmcode, dieser ist vollständig kommentiert.

14.5.4 Kommunikation, FB10

Der Funktionsbaustein FB10 (Kommunikation) dient in erster Linie dazu, Befehle von ZenOn zu empfangen und für die Steuerung innerhalb der SPS sinnvoll zu verarbeiten. Für die interne Verarbeitung werden alle Werte aus Merkern gelesen und in einen globalen DB (DB41) geschrieben. Das Programm ist in 6 Hauptblöcke unterteilt:

- Anforderung eines freien Schlittens
- Abfrage Weichenstellung
- Abfrage Fixierstation
- Transportsystem herunterfahren
- Internes Kommando für Rücksetzung von Ack. 2
- FehlerId- Weitergabe

1. Anforderung eines freien Schlittens

Wenn das Kommando 'K001_SchlittenID' mit dem Wert -1 geschickt wird soll eine neuer Schlitten zu einer Fixierstation angefordert werden. Für Diesen Befehl wird DB41. Neuschlitten auf true gesetzt. Zusätzlich wird auch der Befehl für den Aufruf der Fixierstation in den DB41 geschrieben.

2. Abfrage Weichenstellung

Wird das Kommando 'K001Weiche_stellen' geschickt, springt die Abfrage in den Teil der Weichenstellung. Innerhalb dieser Abfrage werden die Kommandos 'K001_WeichenId', 'K001_Weichenstellung' und 'K001_SchlittenId' abgefragt. Diese Werte werden während des zyklischen Durchlaufs in die zugehörigen Speicherplätze des global DBs gespeichert (DB41).

3. Abfrage Fixierstation

Wird das Kommando 'K002_Fixierst_aktivieren' gesendet, werden alle zugehörigen Befehle in den globalen DB überschrieben (siehe Weichenstellung).

4. Transportsystem herunterfahren

Der Befehl 'K003_TS_herunterfahren' wird genauso wie die vorherigen Befehle behandelt und für die interne Verarbeitung in den globalen DB41 geschrieben.

5. Internes Kommando für Rücksetzung von Ack. 2

In diesem Programmteil wird der Merker 'A002_Befehl_ausgef' abgefragt. Ist dieser auf true, dann setzt der Programmteil die Werte im DB41 auf die Ausgangswerte (false) zurück. Diese Maßnahme ist für die SPS nötig um einen neuen Befehl verarbeiten zu können, da sonst der alte Befehl dauerhaft ansteht. Nach dem zurücksetzen der Werte im DB41, kann ein neuer Befehl ausgeführt werden.

6. FehlerId-Weitergabe

Die Fehler werden über den Merker 'F002_FehlerID' an ZenOn weitergegeben.

14.5.5 Die Weichen,FB60

Um zu steuern, ob ein Schlitten in eines der Nebenbänder einfahren soll, oder auf dem Hauptband verbleibt, gibt es je Nebenband eine pneumatisch gesteuerte Weiche.

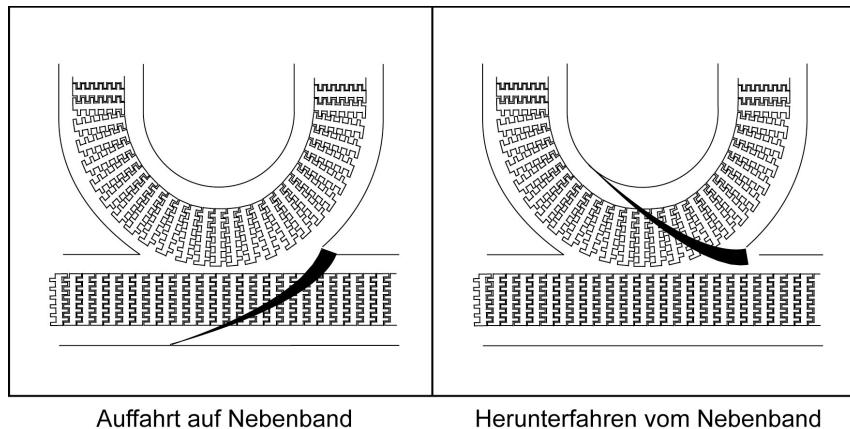


Abbildung 14.25: Stellungen der Weichen

Es werden zwei Arten von Weichen unterschieden:

- Weiche 0

14.5 SPS

Weiche 0 befindet sich zwischen Hauptband und Nebenband 3. Die Weiche befindet sich bei Systemstart in Stellung 0. D.h. wie im Bild Herunterfahren vom Nebenband Abbildung ?? . Soll ein Schlitten von Hauptband auf das Nebenband geschleust werden, wird die jeweilige am DB82 anstehende Schlitten ID mit der geforderten im DB41 verglichen. Kommt der geforderte Schlitten in diesem Bereich, wird er zuerst am Stopper S3 gestoppt. Erst wenn die Weiche geöffnet ist, wird der Stopper eingefahren und der Schlitten kann das Band befahren. Bei dem Sensor B3.3 wird die Weiche Geschlossen und AK 02 gesendet. Wird ein Schlitten vom Nebenband 3 auf das Hauptband geschleust, wird am DB89 die Schlitten ID mit der geforderten verglichen. Kommt der geforderte Schlitten, wird die Weiche nicht geschaltet und der Schlitten verlässt das Nebenband. Erreicht der Schlitten den Sensor B4 so wird AK 02 gesendet. Um eine Kollision hierbei zu vermeiden, wird auf dem Hauptband der Merker 'rechtslinks' bei Sensor B3 gesetzt und bei Sensor B4 rückgesetzt. Ist dieser Merker gesetzt, d. h. der Bereich vor der Weiche ist belegt, wird der Stopper S7 am Nebenband ausgefahren. Erst wenn der Bereich auf dem Hauptband frei ist, darf der Schlitten ausgeschleust werden, und fährt dabei den Stopper S3 am Hauptband aus, um gegebenenfalls ankommende Schlitten zu stoppen, und erst weiterfahren zu lassen wenn ein Mindestabstand gewährleistet ist. Befindet sich ein Schlitten auf Nebenband 3, und es stehen keine Befehle für diesen an, so bleibt dieser immer Kreisend auf dem Nebenband, bis andere Befehle vorliegen. Wenn dieser Schlitten sich nun der Weiche vom Nebenband nähert, wird zuerst der Schlitten an Stopper S7 angehalten. Anhand des Merkers 'rechtslinks' wird überprüft ob der Bereich auf dem Hauptband belegt ist oder nicht. Ist er belegt wird die Weiche nicht geschalten. Erst wenn der Bereich frei ist, schaltet die Weiche und der Stopper S3 am Hauptband wird ausgefahren. Erst wenn die Weiche vollständig geöffnet ist wird der Schlitten gelöst und passiert die Weiche. An Sensor B3.3 wird die Weiche geschlossen und der Stopper am Hauptband eingefahren.

- Weiche1

Weiche 1 befindet sich zwischen Hauptband und Nebenband 2. Die Weiche befindet sich bei Systemstart in Stellung 0. D.h. wie bei Herunterfahren vom Nebenband Abbildung ?? . Soll ein Schlitten von Hauptband auf das Nebenband geschleust werden, wird die jeweilige am DB72 anstehende Schlitten ID mit der geforderten im DB41 verglichen. Kommt der geforderte Schlitten in diesem Bereich, wird er zuerst am Stopper S2 gestoppt. Erst wenn die Weiche geöffnet ist, wird der Stopper eingefahren und der Schlitten kann das Band befahren. Bei Sensor B2.3 wird die Weiche Geschlossen und AK 02 gesendet. Schlitten die dieses Band befahren werden bei B10 'gelöscht'. D.h. ihre Schlitten ID wird entfernt, und sie werden bei Stopper S8 als Leerer Schlitten angehalten.

14.5.6 Die Fixierstationen, FB61

Die Fixierstationen (Abbildung: ?? funktionieren ähnlich wie die Weichen über die DB's der Nummerierung. Alle Fixierstationen arbeiten nach dem gleichen Prinzip. Wenn der Befehl Fixieren gesendet wird, wird dieser über den Kommunikations FB10 in den DB 41 geschrieben. Steht nun bei DB41. Fixierstation_aktivieren = true an, beginnen die weiteren abfragen. Zuerst wird die Fixierstation's ID und die Stellung überprüft. Steht der Befehl für Fixieren an, d.h. DB41.Fixierstellung = true wartet die SPS bis an dem jeweiligen Sensor vor der Fixierstation B2 (Fix1 = DB94, Fix2 = DB96, Fix3 = DB86) die gewünschte Schlitten ID ansteht. Ist dies der Fall, wird der Stopper S2 an der Fixierstation ausgefahren. Steht nun der richtige Schlitten am Stopper S2 (Überprüfung

14.5 SPS

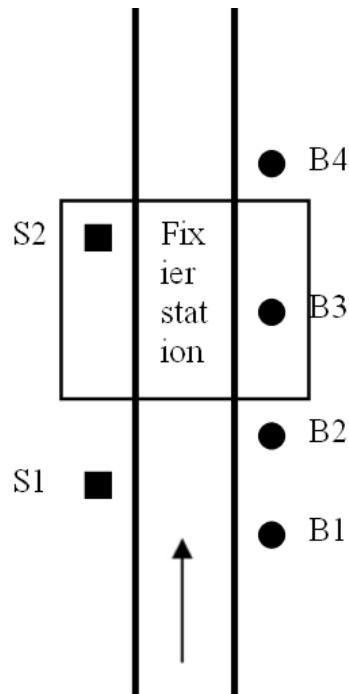


Abbildung 14.26: Schema einer Fixierstation (Sx: Stopper, Bx: Digitaler Sensor)

über: B4, Fix1 = DB96, Fix2 = DB100, Fix3 = DB88) und am Sensor B3 an der Fixierstation, wird diese Ausgefahren und Fixiert den Schlitten. Ist dies Geschehen wird auch AK02 Gesendet.

Auch das Lösen des Schlittens aus der Fixierung, funktioniert bei allen Fixierstationen nach demselben Prinzip. Hierbei steht dann bei DB41.Fixierstellung = false. Es wird überprüft welche Fixierstation gelöst werden soll, und dann der Stopper S2 an der jeweiligen Fixierstation heruntergefahren und die Fixierung gelöst. Ak02 wird jedoch erst gesendet, wenn der gelöste Schlitten den Sensor B4 hinter der jeweiligen Fixierstation überfährt. Dabei wird auch wieder die Schlitten ID überprüft.

14.5.7 Stopperlogik, FB4

Diese Logik steuert die pneumatischen Stopper der FDZ, die über Sensoren angesprochen werden (ohne Fixierstationen). Diese Logik ist im Baustein FB4 realisiert und setzt sich aus verschiedenen Set-Reset-, AND- und OR- Bausteinen zusammen. Über die Schrittstellen wurden verschiedene Ein-, Ausgangs-, und statische Variablen erzeugt, die in der StopperInitialisierung (FB3) für jeden Stopper auf Haupt- und Nebenband verwendet werden. Dieser FB3 Baustein wird von OB1 aufgerufen. Der Datenbaustein DB5 ist hierbei der Instanzdatenbaustein für FB3. Im FB3 werden die einzelnen Stopper über verschiedene DB's angesprochen, die in der Symboltabelle festgelegt sind.

Übergabewerte (Variablen):

- InitiatorSperren:

14.5 SPS

Bei Überfahren dieses Sensors wird ein Eingang über einen SET-RESET Bausstein auf true gesetzt, welches den dahinterliegenden Bereich (bis zum 'freigebenden' Sensor) für alle nachkommenden Schlitten sperrt. Gleichzeitig ist InitiatorSperren der Merker für 'AusgangHoch' oder 'AusgangWechsler'. In einigen Fällen wird neben dem Sensor gleichzeitig abgefragt, ob der nachfolgende Stopper ausgefahren oder eingefahren ist.

- Initiatorschlittenkommt:

Für den Fall dass der Eingang InitiatorSperren gesetzt ist, löst der Initiatorschlittenkommt den Stopper aus.(Stoppt den 2. Schlitten) Falls kein Schlitten Eingang gesetzt hat, ist der Initiatorschlittenkommt ohne Auswirkung.

- Initiatorfreigeben:

Dieser Initiator gibt an, dass ein Schlitten einen bestimmten Bereich wieder verlassen hat. Er setzt den InitiatorSperren des jeweiligen Sensors zurück. Dieser befindet sich in der gesamten Logik immer am Sensor nach dem nachfolgenden Stopper.

- AusgangHoch bzw. AusgangRunter:

Mit diesem Ausgang werden die pneumatischen Stopper mit Zwei-Wege-Ventil geschaltet.

- AusgangWechsler:

Mit diesem Ausgang werden die Ein-Wege-Ventil-Stopper umgeschaltet.

- ZweiWegeVentil:

Mit diesem Eingang wird unterschieden, ob es sich um ein pneumatisches Zwei-Wege-Ventil handelt oder nicht. Ist dies der Fall, werden die Ausgänge AusgangHoch bzw. AusgangRunter angesprochen. Handelt es sich um kein Zwei-Wege-Ventil wird der Ausgang AusgangWechsler verwendet.

- Quellcode:

```
'Stopperlogik'.'DBStopper15'(InitiatorSperren:='Initiator B15.1' , Initiatorfreigeben:= 'Initiator B1.1', Initiatorschlittenkommt:='Initiator B17', AusgangWechsler:= 'Stopper S15', Zwei-WegeVentil:=false);
```

- Erklärung:

Der Schlitten fährt über den Sensor 'B15.1' (InitiatorSperren) und setzt den Eingang auf true (Abbildung: ??).

Nun gibt es zwei Fälle:

1. Der nachfolgende Stopper 1 ist eingefahren, folglich kann der Schlitten auf dem Transportband weiterfahren und löscht den oben.

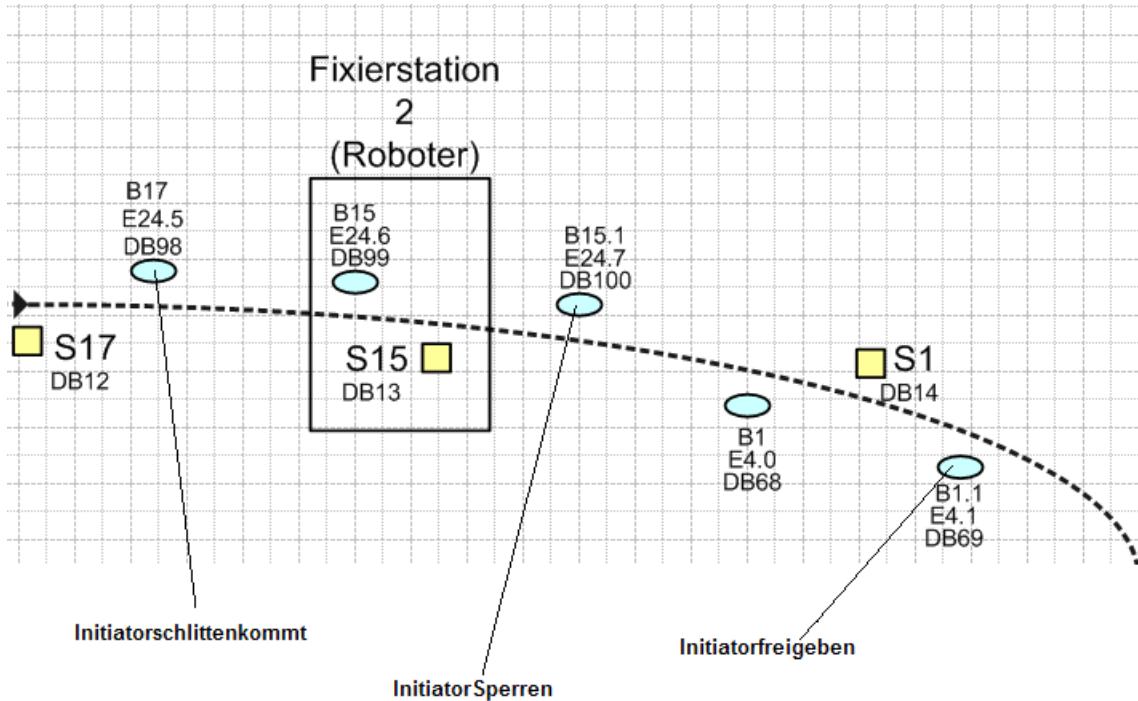


Abbildung 14.27: Beispiel für Stopper 15

- Der nachfolgende Stopper 1 ist ausgefahren, folglich bleibt der Schlitten an Stopper 1 stehen (Abbildung: ??). Eingang für 'InitiatorSperren' bleibt gesetzt. Sobald nun ein weiterer Schlitten den Sensor 'B17' (InitiatorSchlittenkommt) passiert (Abbildung: ??), fährt Stopper 15 aus. Somit wird dieser Schlitten an Stopper S15 angehalten, um eine Kollision mit dem vorhergehenden Schlitten zu vermeiden. Erst wenn der Schlitten an Stopper 1 weiterfahren darf und den Sensor 'B1.1' (InitiatorFreigeben) überfährt (Abbildung: ??), wird Stopper 15 eingefahren und Schlitten2 fährt weiter (Abbildung: ??). genannten Eingang beim überfahren des Sensors 'B1.1' (InitiatorFreigeben).

Beispiel für Stopper 17:

- Quellcode:

```
'Stopperlogik'.DBStopper17'(InitiatorSperren:='DBStopper15'.DX2.0, Initiatorfreigeben:='Initiator B15.1', Initiatorschlittenkommt:='Initiator B19', AusgangWechsler:='Stopper S17', ZweiWegeVentil:=false) ;
```

- Erklärung:

Initiatorfreigeben und Initiatorschlittenkommt funktionieren von der Logik her analog dem Beispiel von Stopper 15. Bei InitiatorSperren wird über 'DBStopper15'.DX2.0 der Stopper 15 abgefragt. InitiatorSperren wird gesetzt, wenn Stopper 15 ausgefahren (.DX2.0) ist.

- Kurvenkollisionsvermeidung:

14.5 SPS

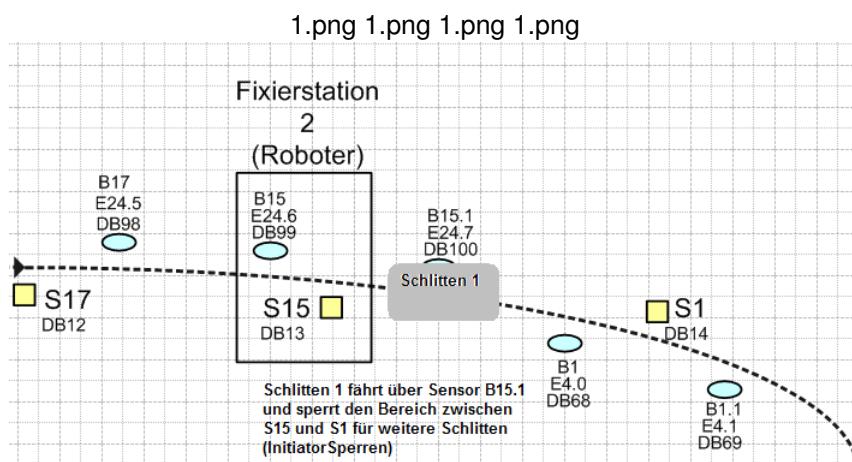


Abbildung 14.28: Beispiel fuer Fall2

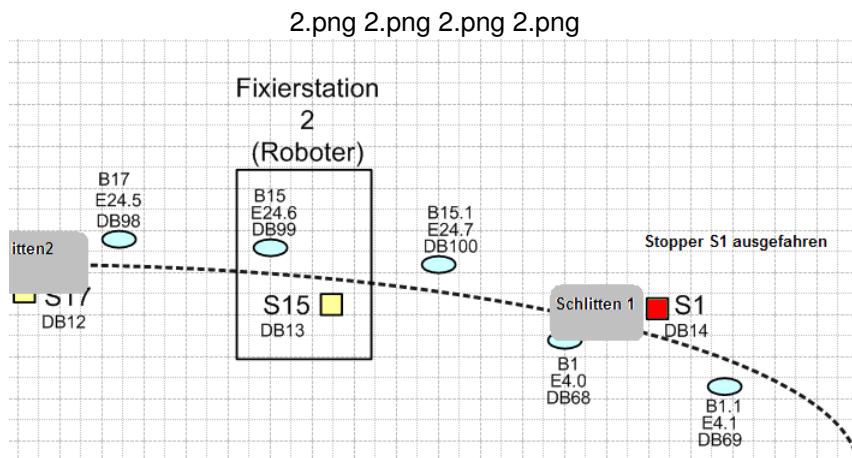


Abbildung 14.29: Beispiel für Fall 2

14.5 SPS

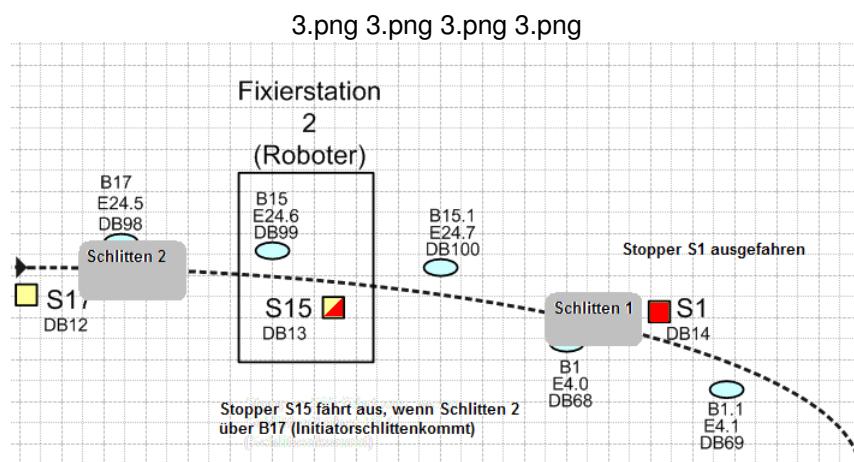


Abbildung 14.30: Beispiel für Fall 2

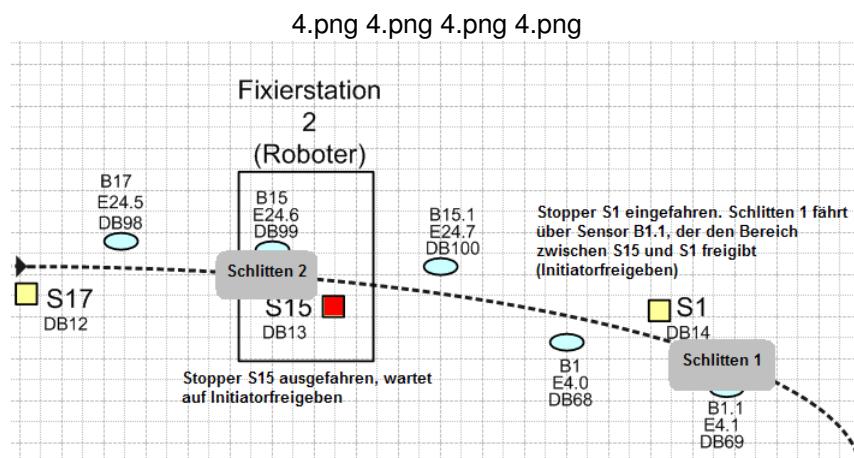


Abbildung 14.31: Beispiel für Fall 2

14.5 SPS

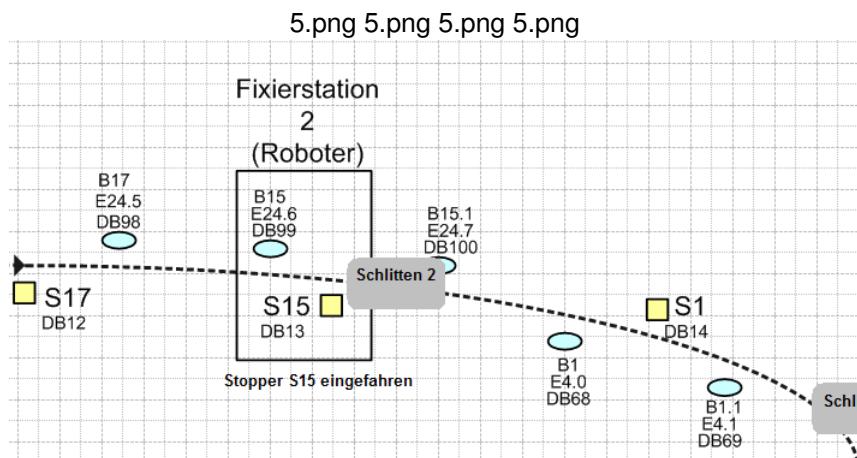


Abbildung 14.32: Beispiel für Fall 2

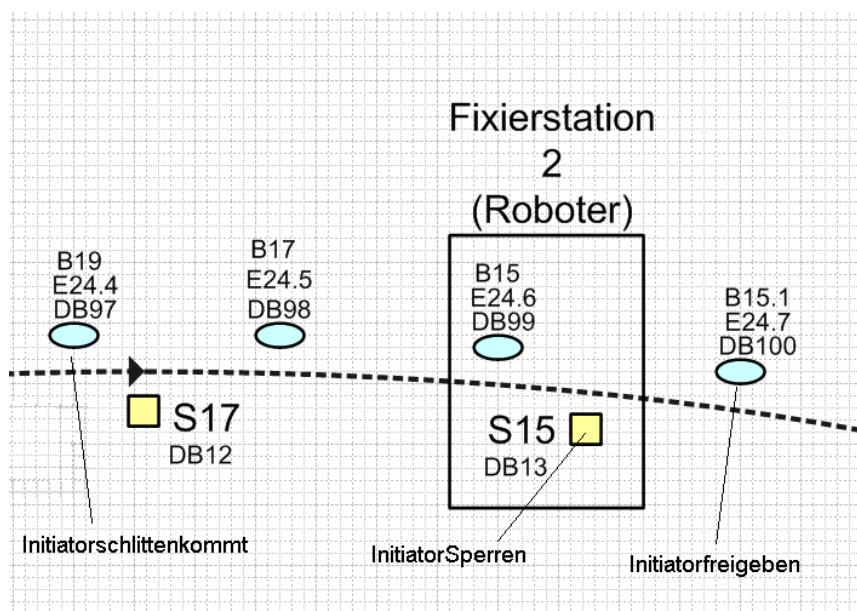


Abbildung 14.33: Beispiel für Stopper 17

14.5 SPS

Fährt ein Schlitten in eine Kurve und überfährt den letzten Sensor in dieser (z.B. 'B1.1'), wird in diesem Fall Stopper 1 ausgefahren, damit eine Kollision in der Kurve vermieden wird.

- Nebenbänder:

Die Stopperlogik auf den Nebenbändern funktioniert analog zum Hauptband.

Belegung der Merker, Funktionsbausteine und Datenbausteine

Symbol	Adresse	Typ	Kommentar
Merker			
Weiche1_aufmachen	M 2.1	BOOL	
Weiche2_aufmachen	M 2.2	BOOL	
Weiche1_zumachen	M 2.3	BOOL	
Weiche2_zumachen	M 2.4	BOOL	
K001_Weiche_stellen	M 40.1	BOOL	
K001_Weichenstellung	M 42.0	BOOL	
rechtslinks2	M 43.2	BOOL	Schlitten steht an Stopper vor Weiche 0
Lösen-1	M 43.3	BOOL	
weiche2	M 43.4	BOOL	
leerschlitten	M 43.7	BOOL	Leerer Schlitten auf Band 2
K002_Fixierst_aktivieren	M 44.1	BOOL	
K002_FixierstStellung	M 46.0	BOOL	
K003_TS_herunterfahren	M 48.0	BOOL	
A001_Befehl_verstanden	M 49.1	BOOL	
A002_Befehl_ausgef	M 51.1	BOOL	
F001_Befehl_n_verstanden	M 52.1	BOOL	
F002_Befehl_n_ausgef	M 53.1	BOOL	
rechtslinks	M 65.0	BOOL	Weichenbereich Weiche 0 belegt
Zustand_weiche	MB 1	BYTE	Test MB
Zustand_fix	MB 3	BYTE	TEST MB
K001_WeichenId	MB 41	BYTE	
K002_Fixierstd	MB 45	BYTE	
A002_SchlittenId	MB 50	BYTE	
F002_FehlerId	MB 54	BYTE	
FehlerbehebungTO	MB 60	BYTE	
K001_SchlittenID	MW 36	INT	
K002_SchlittenID	MW 38	INT	
Funktionsbausteine			
StopperInitialisierung	FB 3	FB 3	
Stopperlogik	FB 4	FB 4	
Transportband	FB 5	FB 5	
Kommunikation	FB 10	FB 10	

14.5 SPS

Symbol	Adresse	Typ	Kommentar
Weichenstellen	FB 60	FB 60	
Fixierstation	FB 61	FB 61	Fixierstation
LeererSchlittengefördert	FB 62	FB 62	
Initiator	FB 101	FB 101	
Nummerierung	FB 102	FB 102	
Weitergabe	FB 103	FB 103	
SchittenID_erzeugen	FC 110	FC 110	
SchittenID_löschen	FC 111	FC 111	
Timeout	FC 112	FC 112	
Datenbausteine			
DBTransportband	DB 3	FB 5	
DBStopperInitialisierung	DB 5	FB 3	
DBStopper2	DB 6	FB 4	
DBStopper3	DB 7	FB 4	
DBStopper4	DB 8	FB 4	
DBStopper11	DB 9	FB 4	
DBStopper16	DB 10	FB 4	
DBStopper13	DB 11	FB 4	
DBStopper17	DB 12	FB 4	
DBStopper15	DB 13	FB 4	
DBStopper1	DB 14	FB 4	
DBStopper9	DB 15	FB 4	
DBStopper5	DB 16	FB 4	
DBStopper7	DB 17	FB 4	
DBStopper10	DB 18	FB 4	
DBStopper6	DB 19	FB 4	
DBStopper8	DB 20	FB 4	
DBZeitmessung	DB 35	FB 35	
GlobalDB-SPS	DB 41	FB 41	
Weiche_stellen	DB 60	FB 60	
Initiator_Daten_B1	DB 68	FB 101	
Initiator_Daten_B1.1	DB 69	FB 101	
Initiator_Daten_B1.4	DB 70	FB 101	
Initiator_Daten_B2	DB 71	FB 101	
Initiator_Daten_B2.4	DB 72	FB 101	
Initiator_Daten_B2.3	DB 73	FB 101	
Initiator_Daten_B10	DB 74	FB 101	
Initiator_Daten_B10.1	DB 75	FB 101	
Initiator_Daten_B6	DB 76	FB 101	
Initiator_Daten_B6.1	DB 77	FB 101	
Initiator_Daten_B8	DB 78	FB 101	
Initiator_Daten_B8.1	DB 79	FB 101	
Initiator_Daten_B2.1	DB 80	FB 101	
Initiator_Daten_B3	DB 81	FB 101	

14.5 SPS

Symbol	Adresse	Typ	Kommentar
Initiator_Daten_B3.4	DB 82	FB 101	
Initiator_Daten_B3.3	DB 83	FB 101	
Initiator_Daten_B9	DB 84	FB 101	
Initiator_Daten_B9.1	DB 85	FB 101	
Initiator_Daten_B14	DB 86	FB 101	
Initiator_Daten_B5	DB 87	FB 101	
Initiator_Daten_B7	DB 88	FB 101	
Initiator_Daten_B5.1	DB 89	FB 101	
Initiator_Daten_B4	DB 90	FB 101	
Initiator_Daten_B4.1	DB 91	FB 101	
Initiator_Daten_B11.2	DB 92	FB 101	
Initiator_Daten_B18	DB 93	FB 101	
Initiator_Daten_B18.1	DB 94	FB 101	
Initiator_Daten_B16	DB 95	FB 101	
Initiator_Daten_B16.1	DB 96	FB 101	
Initiator_Daten_B19	DB 97	FB 101	
Initiator_Daten_B17	DB 98	FB 101	
Initiator_Daten_B15	DB 99	FB 101	
Initiator_Daten_B15.1	DB 100	FB 101	
DB_Nummerierung	DB 102	FB 102	
DBWeitergabe	DB 103	FB 103	
KommunikationZenOn	DB 120	FB 120	

15 EAsystem

15.1 Aufbau / Struktur der Software

Die E/A-Station-Steuerungsoftware ist in 3 Packages eingeteilt: der „default package“, „net“ und „fdzFunktionssimulator.presentationLayer“. In der „default package“ befindet sich der für die EAStation wichtigste Code für die Befehlshandhabung anhand der GUI und der Erstellung bzw. Abarbeitung des Befehlsstrings in Verbindung mit der Controlschicht. Das „net“-Package ist hauptsächlich dafür da, eine TCP/IP Verbindung mit der Controlschicht zu schaffen, auf welcher die vorhin erwähnten Befehlsstrings transferiert werden. Das „fdzFunktionssimulator.presentationLayer“ dient lediglich zur Nutzung des SmartieLayoutDialog aus dem Funktionsimulator.

15.1.1 Klassendiagramm

Das folgende Klassendiagramm stellt das ganze Projekt mit ihren Klassen und Abhangigkeiten dar. Die Klassen sind hier fur die bersichtlichkeit ohne Attribute, Operationen oder Unterklassen dargestellt.

Es lässt sich eine grobe Einteilung in die Commands (links), der GUI (rechts) und der Connection (links unten) erkennen.

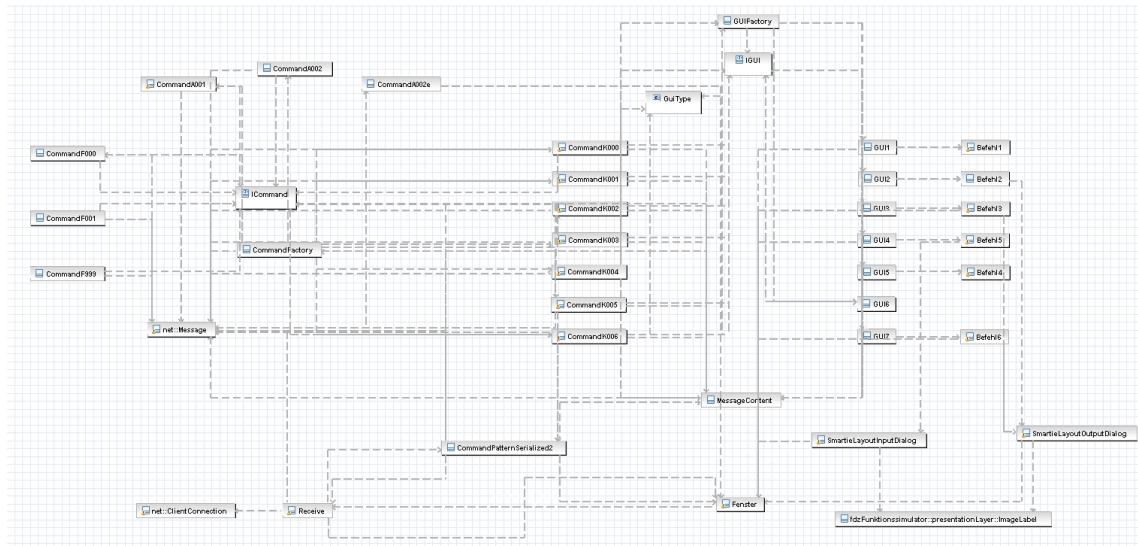


Abbildung 15.1: Klassendiagramm des ganzen EASteuerungsprojekts

15.1.2 Klassen und Interfaces

- Fenster

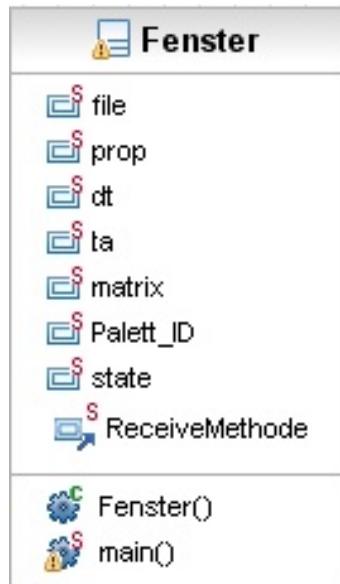


Abbildung 15.2: Klasse Fenster

In der Fensterklasse befindet sich die main-Funktion.

Hier wird die Startseite der GUI und deren Parameter erstellt. Hier befinden sich auch „ActionListener“, die je nach Art des gewünschten Befehls reagieren, und wenn nötig ein PoPUp öffnen. Außerdem wurde beim Start die Möglichkeit der IP Eingabe realisiert.

- CommandPatternSerialized2

15.1 Aufbau / Struktur der Software

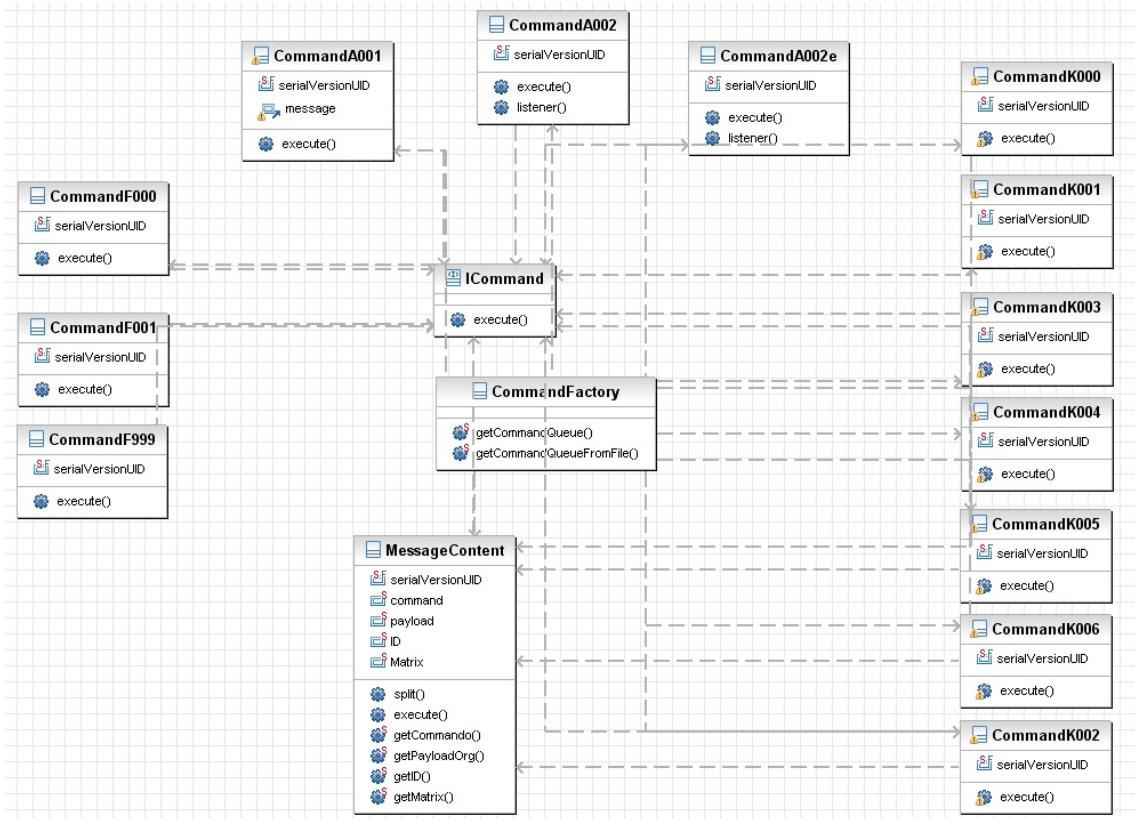


Abbildung 15.3: Klassendiagramm CommandPatternSerialized2

Das CommandPattern erzeugt viele Unterklassen. Diese wiederum dienen der Befehlsabarbeitung der EAStation.

Zuerst wird aber die Ausgabe, dass die Queue gestartet ist an die Kommandozeile übergeben bis diese durchgelaufen ist und das Ende der Queue angezeigt wird. Mit der Methode „runCommands“ werden alle Befehle in der Queue ausgeführt. Zusätzlich findet hier die Realisierung der Recovery statt.

– CommandFactory



Abbildung 15.4: Klasse CommandFactory

CommandFactory generiert die passende Queue fuer ein gegebenes Szenario.

– ICommand

15.1 Aufbau / Struktur der Software



Abbildung 15.5: Klasse ICommand

Sorgt das der nächste Befehl aus der Queue ausgeführt wird.

- CommandA001



Abbildung 15.6: Klasse CommandA001

Gibt das Ack1 in der Form „A001: Befehl verstanden“ aus. Zeigt desweiteren auch den Befehlsstring „message“ an.

- CommandA002



Abbildung 15.7: Klasse CommandA002

Gibt das Ack2 in der Form „A002: Befehl ausgeführt“ aus. Zeigt desweiteren auch den Befehlsstring „message“ an.

- CommandA002e

15.1 Aufbau / Struktur der Software



Abbildung 15.8: Klasse CommandA002e

Gibt das Ack2 in der Form „A002e: Befehl ausgeführt“ aus. Zeigt desweiteren auch den Befehlsstring „message“ an. Zusätzlich wird hier die PalettenID mit der jeweiligen Farbauswahl als String ausgegeben.

- CommandK000



Abbildung 15.9: Klasse CommandK000

Gibt die befehlsspezifischen Infos und Anweisungen aus. Bestimmt den GUI-Type und gibt den Befehl zum Öffnen des jeweiligen GUI-Fensters.

- CommandK001



Abbildung 15.10: Klasse CommandK001

Gibt die befehlsspezifischen Infos und Anweisungen aus. Bestimmt den GUI-Type und gibt den Befehl zum Öffnen des jeweiligen GUI-Fensters.

- CommandK002

15.1 Aufbau / Struktur der Software



Abbildung 15.11: Klasse CommandK002

Gibt die befehlsspezifischen Infos und Anweisungen aus. Bestimmt den GUI-Type und gibt den Befehl zum Öffnen des jeweiligen GUI-Fensters.

- CommandK003



Abbildung 15.12: Klasse CommandK003

Gibt die befehlsspezifischen Infos und Anweisungen aus. Bestimmt den GUI-Type und gibt den Befehl zum Öffnen des jeweiligen GUI-Fensters.

- CommandK004



Abbildung 15.13: Klasse CommandK004

Gibt die befehlsspezifischen Infos und Anweisungen aus. Bestimmt den GUI-Type und gibt den Befehl zum Öffnen des jeweiligen GUI-Fensters. LöschH ordnungsgemäß die Queue.

- CommandK005

15.1 Aufbau / Struktur der Software



Abbildung 15.14: Klasse CommandK005

Gibt die befehlsspezifischen Infos und Anweisungen aus. Bestimmt den GUI-Type und gibt den Befehl zum Öffnen des jeweiligen GUI-Fensters.

- CommandK006



Abbildung 15.15: Klasse CommandK006

Gibt die befehlsspezifischen Infos und Anweisungen aus. Bestimmt den GUI-Type und gibt den Befehl zum Öffnen des jeweiligen GUI-Fensters.

- CommandF000



Abbildung 15.16: Klasse CommandF000

Gibt Fehlermeldung „F000: Befehl wurde nicht verstanden“ aus. Zeigt desweiteren auch den Befehlsstring „message“ an.

- CommandF001

15.1 Aufbau / Struktur der Software



Abbildung 15.17: Klasse CommandF001

Gibt Fehlermeldung „F001: Befehl wurde nichT ausgeführt“ aus. Zeigt desweiteren auch den Befehlsstring „message“ an.

- CommandF999



Abbildung 15.18: Klasse CommandF999

Gibt Fehlermeldung „F999: E/A Sytem kann nichT weiterarbeiten“ aus. Zeigt desweiteren auch den Befehlsstring „message“ an.

- MessageContent

15.1 Aufbau / Struktur der Software



Abbildung 15.19: Klasse MessageContent

Klasse um den Inhalt der empfangenen Nachricht abzufangen, bevor dieser von Ack1 überschrieben wird.

- GUI

15.1 Aufbau / Struktur der Software

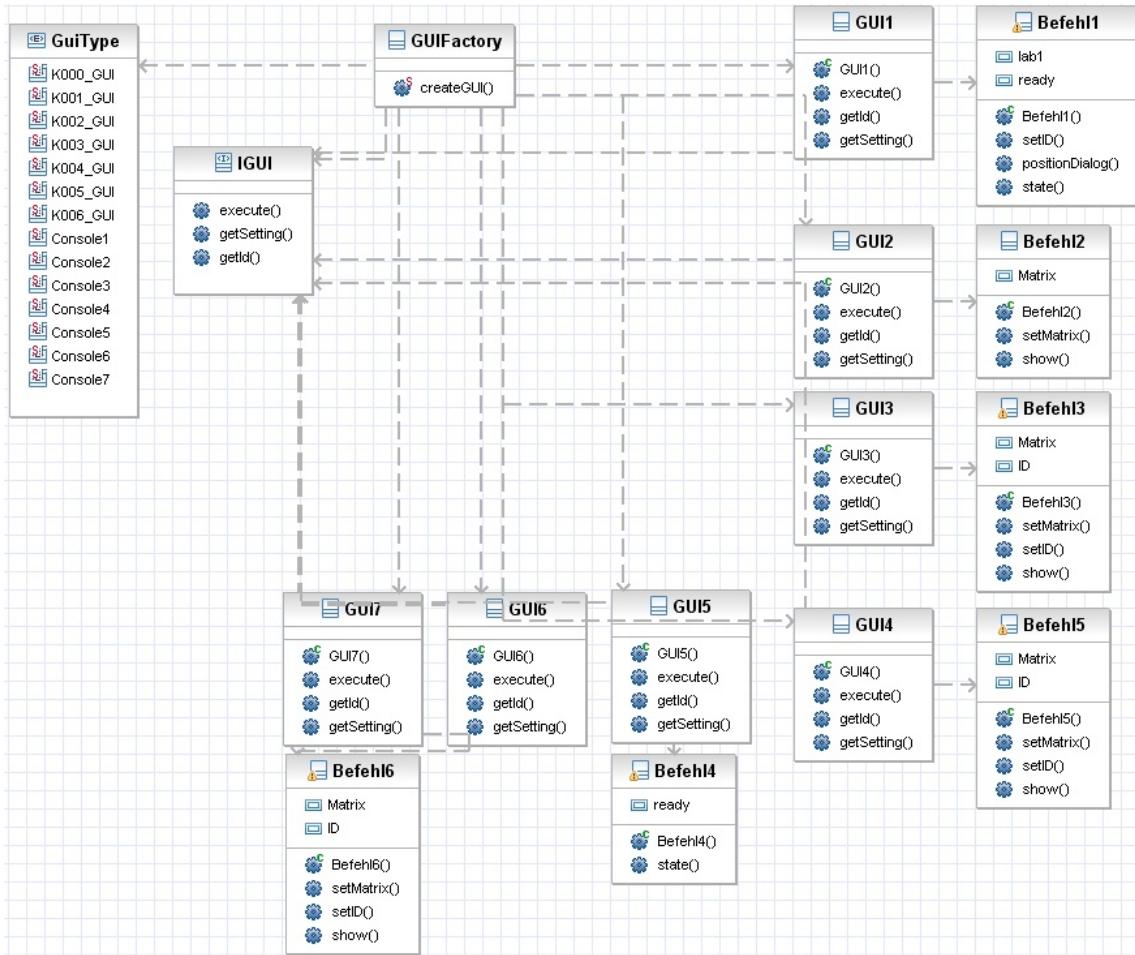


Abbildung 15.20: Klassendiagramm GUI

In diesem Klassenkonstrukt wird die Realisierung der Anzeige und Funktionalität der Fenster erledigt.

- **GUIFactory**



Abbildung 15.21: Klasse GUIFactory

GUIFactory generiert die passende GUI für den entsprechenden GUI-Type.

15.1 Aufbau / Struktur der Software

- IGUI



Abbildung 15.22: Klasse IGUI

Sorgt das der nächste Befehl aus der Queue ausgeführt wird. Deklariert den String „getID“ und „getSetting“ welche die GUI's 1-7 nutzen.

- GUIType

15.1 Aufbau / Struktur der Software



Abbildung 15.23: Klasse GuiType

Definiert die möglichen GUI-Types.

- GUI1

15.1 Aufbau / Struktur der Software



Abbildung 15.24: Klasse GUI1

Ruft die Klasse „Befehl1“ mit den entsprechenden Parametern auf.

- Befehl1



Abbildung 15.25: Klasse Befehl1

Beinhaltet die ganze Funktionalität und Design des Fensters.

- GUI2

15.1 Aufbau / Struktur der Software

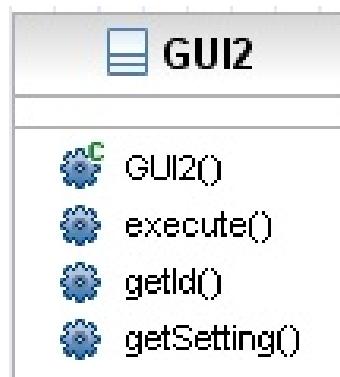


Abbildung 15.26: Klasse GUI2

Ruft die Klasse „Befehl2“ mit den entsprechenden Parametern auf.

- Befehl2



Abbildung 15.27: Klasse Befehl2

Beinhaltet die ganze Funktionalität und Design des Fensters.

- GUI3

15.1 Aufbau / Struktur der Software



Abbildung 15.28: Klasse GUI3

Ruft die Klasse „Befehl3“ mit den entsprechenden Parametern auf.

- Befehl3



Abbildung 15.29: Klasse Befehl3

Beinhaltet die ganze Funktionalität und Design des Fensters.

- GUI4

15.1 Aufbau / Struktur der Software



Abbildung 15.30: Klasse GUI4

Ruft die Klasse „Befehl5“ mit den entsprechenden Parametern auf.

- Befehl5



Abbildung 15.31: Klasse Befehl5

Beinhaltet die ganze Funktionalität und Design des Fensters.

- GUI5

15.1 Aufbau / Struktur der Software



Abbildung 15.32: Klasse GUI5

Ruft die Klasse „Befehl4“ mit den entsprechenden Parametern auf.

- Befehl4



Abbildung 15.33: Klasse Befehl4

Beinhaltet die ganze Funktionalität und Design des Fensters.

- GUI6

15.1 Aufbau / Struktur der Software



Abbildung 15.34: Klasse GUI6

Gibt Meldung „System wird heruntergefahren“ aus.

- GUI7

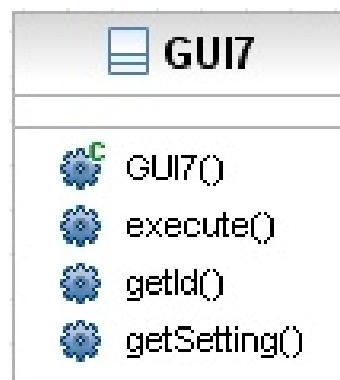


Abbildung 15.35: Klasse GUI7

Ruft die Klasse „Befehl6“ mit den entsprechenden Parametern auf.

- Befehl6

15.1 Aufbau / Struktur der Software



Abbildung 15.36: Klasse Befehl6

Beinhaltet die ganze Funktionalität und Design des Fensters.

16 zenOn 6.20

16.1 Implementierung der zenOn-Schnittstelle

Die Kommunikation mit der SPS funktioniert über das Programm **zenOn 6.20 SP4** von der Firma COPA-DATA. Das Programm ist in der Lage, Kommandos an die SPS zu senden und Aktionen von der SPS zu interpretieren, damit diese benutzerfreundlich angezeigt werden können (weitere Infos hierzu im Kapitel 'GUI').

COPA-DATA hat für zenOn eine Programmierschnittstelle bereitgestellt, die es ermöglicht, den Ablauf per C++ zu überwachen und zu steuern. Diese Schnittstelle ist über ActiveX ansprechbar. Damit das Interface startet und funktioniert, muss die zenOn Runtime mit dem gewünschten Projekt aktiv sein, weil die Schnittstelle versucht, auf die Runtime zu verbinden, und automatisch das aktive Projekt einliest.

16.2 Implementierung der GUI

16.2 Implementierung der GUI

17 zenOn Proxy

17.1 Anforderungsanalyse

Aufgabe des Zenon-Proxy ist es die Kommunikation zwischen Steuerung und SPS zu implementieren. Der Datenaustausch ist hierbei durch die Schnittstellen der jeweiligen Kommunikationspartner bestimmt. Die Anweisungen der Steuerung sind String-basierend, da zur Übertragung das für Ethernet typische Protokoll TCP/IP verwendet wird. Die übers Netzwerk gesendeten Befehle können in zwei Teile unterteilt werden, den Header mit den für die Verbindung notwendigen und den Payload mit den nachrichtenspezifischen Informationen. Nachdem Empfangen einer Nachricht wird diese in die einzelnen Informationsgehalte zerteilt und mittels COM-Schnittstelle auf der SPS gesetzt. Im Anschluss an die Übertragung eines Befehls muss eine Reaktion der SPS erfolgen, bevor der nächste Befehl gesendet werden kann. Hierzu kommen zwei Acknowledges zum Einsatz, die mit Hilfe von Merkern implementiert werden können. Diese Merker werden Zyklisch durch den Zenon-Proxy abgefragt, bis durch Wertänderung der ordnungsgemäß Erhalt der Nachricht (Acknowledge 1) beziehungsweise die erfolgreiche Abarbeitung des Befehls (Acknowledge 2) bescheinigt wird. Nachdem erfolgreichen Empfang von Acknowledge 2 werden die Parameter der SPS wieder zurückgesetzt, damit der fehlerfreie Empfang einer weiteren Nachricht möglich ist. Eine weitere Anforderung neben dem Übersetzen und Durchreichen von Steuerbefehlen ist die Gewährleistung einer verlustfreien Datenübertragung. Für die Implementierung des Zenon-Proxies bedeutet dies, dass Befehle trotz eventueller Systemabstürze beziehungsweise benutzerbedingten Fehlanwendungen ordnungsgemäß abgearbeitet werden können. Diese Problematik wird durch die Verwendung eines Recovery-Files gelöst, der zu jedem Zeitpunkt des Programmablaufs Speichert, welcher Teilbefehl zuletzt ordnungsgemäß durchgeführt wurde und welcher Schritt als nächstes abgearbeitet werden muss.

17.2 Laden und Kompilieren der zenon-Proxy Projektdatei

Der Zenon-Proxy wurde in der Programmierumgebung **VisualStudio8 Professional** implementiert. Als Programmiergerüst wurde das **Microsoft .Net Framework der Version 3.5** verwendet. Wie alle Programmdateien der FDZ befindet sich auch der Projektordner **fdzZENON-ProxySS2009** des Zenon-Proxies auf dem SVN-Server unter dem Ordner **/scr**. Zum Starten des Projekts ist die VisualStudio Projektdatei **Proxy.sln** auszuwählen, welche sich in der obersten Ebene des Projektordners befindet. Nach dem Laden der Projektdatei inklusive sämtlicher Klassen und Verweise kann der Proxy kompiliert werden. Hierzu kann entweder der grüne Play-Button gedrückt oder die Shortcut-Funktion F5 betätigt werden. Zum Ändern der jeweiligen Klassen ist am Rechten Rand der Programmierumgebung eine Klassenansicht aufgeführt, in der die zu ändernden Klassen aufgeführt sind.

17.3 Die Main Methode

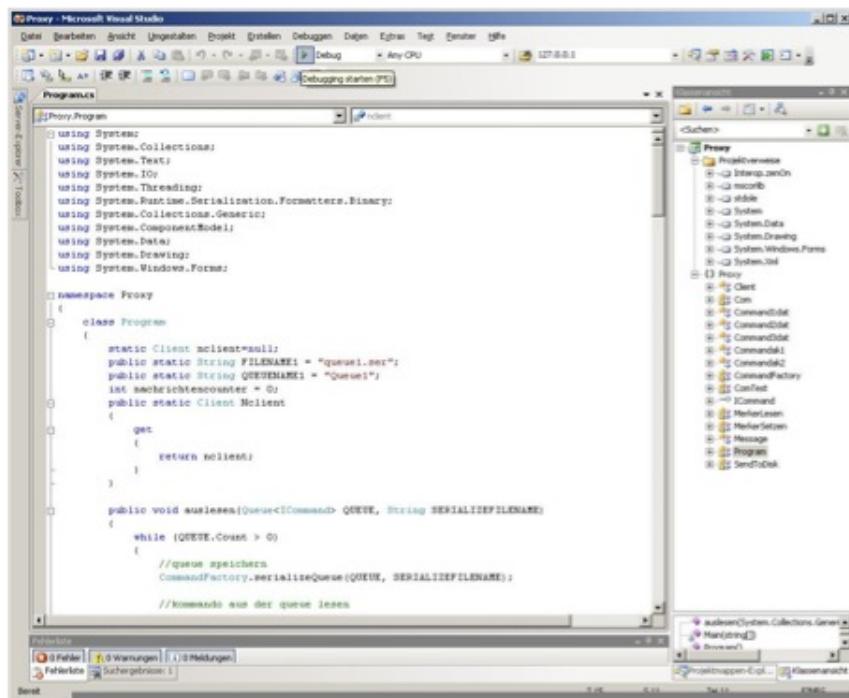


Abbildung 17.1: Konfiguration der Verbindungsparameter

17.3 Die Main Methode

17.3.1 Settings und Versionen

Durch die Erzeugung der Objekte

```
Settings s = new Settings();
Version version = new Version();
```

kann im folgendem die Ip-Adresse sowie der Port des Servers festgelegt werden.

17.3.2 Der Programm Start

Nachdem die Ip-Adresse sowie der Port festgelegt sind wird durch die Instanziierung des Objektes

```
Program p = new Program()
```

der Programmablauf gestartet.

17.4 Programmbeschreibung zenOn-Proxy

17.4.1 Die Klasse Programm

Die Methode [runCommands(QUEUEUNAME1, FILENAME1)] wird im Konstruktor der Klasse Programm aufgerufen. Sie bekommt als übergabe Parameter den Queuenamen (QUEUEUNAME1) sowie den Dateinamen (FILENAME1) für das speichern der Queue übergeben. Nun wird die Queue als Generischer Datentyp des Interfaces ICommand angelegt. Im folgendem wird die Methode folder() aufgerufen welche den Pfad der EigenenDateien zurück liefert um diesen später für das Speichern der Queue sowie der temporären Datei zu verwenden. Nachdem die Pfade nun richtig zusammengesetzt sind wird nun geprüft ob ein Recoveryfall vorliegt. Dazu wird die Methode [recoveryFileExists(laufwerk) — recoveryFileExists(recoverlaufwerk)] benötigt diese prüft ob in dem übergebenen Pfad die Datei exsistiert. Sie gibt true für exsistenz und false für nicht vorhanden zurück.

In der folgenden if-Abfrage [if (recoveryFileExists(recoverlaufwerk) && recoveryFileExists(laufwerk))] wird geprüft ob evtl. beide Recoverydateien vorliegen. Sollte diese der Fall sein war ein Absturz nachdem die fertig Queue gespeichert wurde jedoch bevor das tempfile gelöscht wurde der Grund. In diesem Fall wird zunächst das tempfile mittels dem bereits angelegtem Objekt der Klasse [SendToDisk std = new SendToDisk();] gelöscht dazu wird folgender Methodenaufruf benutzt [std.delete tempfile(laufwerk);]. Sollten nicht beide Dateien exsistieren wird mittels den beiden if-Abfragen

[if (recoveryFileExists(laufwerk))] und

[if (recoveryFileExists(recoverlaufwerk))] bestimmt an welchem Punkt es zum Programmabsturz kam.

Abschließend wird ein Objekt des Interfaces ICommand erzeugt um das letzte Objekt aus der Queue anzeigen zu können. Dies geschieht mittels der Systemmethode [Peek()]. Das Element der Queue wird nun ausgegeben um im Zweifel entscheiden zu können ob der letzte Befehl als Recoverfall ausgeführt werden soll.

Nun kann der Benutzer entscheiden ob er die Recoverydatei laden möchte oder nicht.

Wenn sich der Benutzer für das Ausführen von Recovery entscheidet wird zunächst geprüft ob bereits eine Verbindung zum Server besteht. Sollte dies nicht der Fall sein wird entsprechend der in der main – Methode eingegebenen Ip-Adresse eine Verbindung aufgebaut.

Nun wird nochmals wie oben beschrieben geprüft welche Art von Recovery vorliegt.

Solle die erste if- Abfrage if (recoveryFileExists(recoverlaufwerk)) true zurückliefern wird der Pfad der Recoverdatei angezeigt und mittels dem Aufruf [queue = CommandFactory.getCommandQueueFromFile(recoverlaufwerk);] das Recoverfile aus der Datei geladen und in eine Queue übersetzt. Anschließend wird die durch den Aufruf der Methode [public void auslesen(Queue<ICommand> QUEUE, String SERIALIZEFILENAME)] die sich noch in der Queue befindlichen Befehle ausgeführt. Abschließend wird jetzt ein neue Instanz von Programm erzeugt.

17.4 Programmbeschreibung zenOn-Proxy

Sollte die Abfrage [if (recoveryFileExists(laufwerk))] true zurückliefern so wird mittels Message message = std.getMessageFromFile(laufwerk);] die in der Klasse SendToDisk von der Festplatte zurückgelesene Message eingelesen. Im Anschluss wird die Instanz von Message der statischen Methode[queue = CommandFactory.getCommandQueue(QUEUENAME, message);] übergeben welche wiederum eine Queue zurückgibt. Im Folgendem wird die Queue wie im ersten Beispiel bereits beschrieben abgehandelt. Auch hier wird am Ende des Recoveryfalls [Program programm = new Program();] aufgerufen um das Programm erneut zu starten.

Sollte sich der Benutzer gegen Recovery entscheiden so werden im else- Zweig beide Recoverdatein durch [SendToDisk.deletequeue(recoverlaufwerk);] und [SendToDisk.deletequeue(laufwerk);] gelöscht.

Sollte kein Recoveryfall vorhanden sein so wird in der äußeren else Verzweigung der Speicher für eine neue Nachricht angelegt danach wird geprüft ob eine Verbindung existiert sollte keine exsistieren so wird eine neue Verbindung aufgebaut.

Nun wird der normale Programmablauf gestartet dazu wird eine endlos polling Schleife gestartet. Innerhalb der Schleife wird nun eine neue Instanz der Klasse Message erstellt. Sollten bedingt durch einen Recoverfall im Tcp/Ip – Buffer nicht ausgelesene Bytes verbleiben so wird nun mittels [nclient.clearBuffer();] der Buffer geleert. Anschließend wird mittels [nachricht.readmassage()] eine Nachricht empfangen und ausgelesen. Da die nachricht nun im System vorhanden ist wird sie nun zur Ausfallsicherheit mittels [std.makefile(nachricht, laufwerk);] auf die Festplatte geschrieben. Im Anschluss wird die Nachricht mittels [queue = CommandFactory.getCommandQueue(QUEUENAME, nachricht);] zu einer Queue zusammengesetzt und abschließend durch [auslesen(queue, recoverlaufwerk);] ausgeführt.

17.4.2 Die Klasse Client

Innerhalb der Klasse Client wird zunächst ein neuer socket erzeugt anschließend werden die Variablenservername und port für den späteren Verbindungsaufbau initialisiert.

17.4.2.1 Methoden der Klasse Client

- public void connectToServer(String SERVERNAME, int PORT)

Der Methode connectToServer wird der Servername sowie der Port aus der Klasse Programm übergeben. Im Anschluss wird innerhalb einer while – Schleife die Verbindung zum Server aufgebaut. Sollte eine FormatException auftreten so wird das Programm nach Ausgabe der entsprechenden Exception beendet. Sollte eine InvalidOperationException auftreten so wird der bestehende Socket geschlossen anschließend wird ein neuer Tcp/Ip Socket geöffnet.

- public byte[] datenempfangen(int size)

In dieser Methode werden durch den geöffneten Socket Daten empfangen. Hierzu wird der Methode mittels dem Übergabeparameter size die Länge der zu empfangenden Daten übergeben da im Programmablauf zuerst der Header und anschließend der Payload ausgelesen wird. Innerhalb der folgenden while – Schleife wird empfangen bis die Anzahl der

17.4 Programmbeschreibung zenOn-Proxy

empfangenen Bytes mit der in size übergebenen Anzahl übereinstimmt. Dies ist notwendig da das Tcp/Ip – Protokoll nicht gewährleistet, dass die Daten unfragmentiert übermittelt werden.

- public void sendToServer(string SENDMESSAGE, int size)

Mittels dieser Methode werden evtl. entstandene Fehler zum Server übermittelt. Auch hier muss mittels size die Länge der zu sendeten Daten angegeben werden. Wie bereits in der Methode datenempfangen() beschrieben muss auch hier innerhalb einer while-Schleife gesendet werden um die vollständige Datenübertragung zu gewährleisten.

- public byte[] StrToByteArray(string STR)

In dieser Methode wird ein übergebener String für das Senden eines Fehlers an der Server zurück in ASCII codiert. Sie gibt ein entsprechendes Byte Array zurück.

- public void clearBuffer()

Falls Daten z.B. bedingt durch einen Absturz im Tcp/Ip Buffer verbleiben wird dieser hier geleert. Hierzu wird ein Array mit der maximalen Tcp/Ip Protokollgröße angelegt und anschließend die mit socket.Available bestimmte im Buffer verbliebene Datengröße abgeholt.

17.4.3 Die Klasse Message

In der Klasse Message wird die Nachricht über das Netzwerk eingelesen und in Ihre einzelnen Bestandteile zerlegt um sie im folgenden weiter verarbeiten zu können.

17.4.3.1 Die Methoden der Klasse Message

- public int getLength()

Hier wird die Länge der Nachricht bestimmt. Sollte eine Exception auftreten so wird der Receivebuffer geleert und der Fehlercode „999“ für Exception beim Auslesen zurückgegeben.

- public void readmassage()

In der Methode readmassage() wird das Empfangen der Daten gestartet mittels dem Aufruf [c = Program.Nclient.datenempfangen(25)] wird der Header der Nachricht in ein Bytarray der Klasse eingelesen. Anschließend wird das Byte Array in einen String Encodiert und die Nachricht in empfänger, sender, messageid, nachrichtentyp und befehlsnummer zerlegt. Anschließend wird mittels dem Aufruf [getLength();] die Länge der Nachricht bestimmt. Sollte die zurückgegebene Länge 29 sein so werden im Folgendem die Nutzdaten eingelesen, ebenfalls encodiert und anschließend in die entsprechenden id's zerlegt. Sollte eine FormatException auftreten so wird die gespeicherte Message gelöscht anschließend ein Fehler generiert und mittels des bestehenden socket an den Server gesendet. Nun wird der empfangsbuffer geleert und eine neue Nachricht eingelesen. Sollte die Nachricht nicht 29 Bytes lange haben, so wird zunächst geprüft ob evtl. der Befehl K003 „Herunterfahren“ vorliegt. Sollte dies der Fall sein so werden alle Id's auf Null gesetzt. Wenn die

17.4 Programmbeschreibung zenOn-Proxy

Naricht nicht 29 Bytes umfasst und nicht K003 sein sollte so wird wie bereits beschrieben ein Fehler gesendet und anschließend eine neue nachricht eingelesen.

17.4.4 Die Klasse CommandFactory

Innerhalb dieser Klasse wird die Queue für die verschiedenen Befehle zusammengesetzt und gespeichert. Desweiteren wird diese Klasse für das Laden einer bereits gespeicherten Queue benutzt.

17.4.4.1 Die Methoden der Klasse CommandFactory

- public static Queue< ICommand> getCommandQueue(String QUEUENAME, Message MESSAGE)

Hier wird zunächst eine Queue instanziert anschließend wird die Queue entsprechend des Nachrichtentyps zusammengesetzt. Hierzu werden Instanzen der entsprechenden Klassen in der Queue erzeugt.

- public static void serializeQueue(Queue< ICommand> QUEUE, String PATH)

In dieser Methode wird die übergebene Queue im übergebenen Pfad auf der Festplatte für den Recoverfall gespeichert. Das vorher abgespeicherte File wird im Anschluss gelöscht.

- public static Queue< ICommand> getCommandQueueFromFile(String SERIALIZEFILENAME)

Durch die Deserialisierung der sich auf der Festplatte befindlichen Datei kann hier die Queue bei einem Recoverfall wieder geladen werden.

17.4.5 Die Klasse Command1dat

Die Klasse public class Command1dat : ICommand ist eine abstrakte Klasse der Schnittstelle ICommand das heißt, sie wird vom ICommand Interface abgeleitet. Initialisiert wird die Klasse Command1dat beim Erstellen der Queue in der Klasse CommandFactory.

- Der Konstruktor Command1dat

Der Konstruktor internal Command1dat(Message MESSAGE) bekommt ein Objekt MESSAGE der Klasse Message übergeben, damit in der Klasse Command1dat auf die Variablen der gesendeten Nachricht zugegriffen werden kann.

- Die Methode getMessage()

Die Methode public string getMessage() gibt den Nachrichtentyp (z.B. "K001") als String zurück. Diese Methode wird bei einem Recoveryfall vor der Abfrage zur Ausführung der Recoverydatei durch die Klasse Programm aufgerufen, damit der Benutzer weiß welcher Befehl nicht ordnungsgemäß abgearbeitet wurde.

17.4 Programmbeschreibung zenOn-Proxy

- Die execute() Methode

Die public void execute() Methode führt den eigentlichen Arbeitsschritt nach dem Empfang der Nachricht aus, nämlich das Setzen der Merker auf der SPS. Hierzu wird eine Instanz der Klasse MerkerSetzen erzeugt. Anschließend wird die Methode setzen() aufgerufen, welche die zu schreibenden Informationen übergeben bekommt.

17.4.6 Die Klasse MerkerSetzen

Wie bereits beschrieben dient die Klasse MerkerSetzen zum setzen der Merker auf der SPS. Hierzu wird die setzen() Methode der Klasse verwendet, welche die aus der Nachricht extrahierten Informationen auf die Merker der SPS schreibt.

17.4.6.1 Die setzen() Methode

Die Methode public void setzen(string nachrichtentyp, int id1, int id2, int id3) bekommt neben den Nachrichtentyp (z.B. "K001") auch drei Ids der Nachricht übergeben. Der Nachrichtentyp dient hierbei zum Identifizieren der empfangenen Nachricht, um anschließend die passende Methode zum setzen der Merker aufrufen zu können. Das Setzen der Variable erfolgt über die COM-Schnittstelle von Zenon, welche am Anfang der Methode initialisiert wird. Um welchen Nachrichtentyp es sich handelt wird hierbei durch eine switch-case-Anweisung ermittelt, die in den case-Blöcken durch Aufruf der zugehörigen Write-Methoden die Nachrichten-Ids dem Nachrichtentyp entsprechend interpretiert.

17.4.7 Die Commandak1 Klasse

Auch die public class Commandak1 : ICommand ist eine abstrakte Klasse der Schnittstelle ICommand. Die Initialisierung erfolgt hierbei analog zur Klasse Command1dat über die CommandFactory. Aufgabe der Klasse ist es die mit Acknowledge1 in Verbindung stehende Kommunikation abzuwickeln.

- Der Konstruktor Commandak1

Der Konstruktor internal Commandak1(Message MESSAGE) bekommt ein Objekt MESSAGE der Klasse Message übergeben, damit in der Klasse Commandak1 auf die Variablen der gesendeten Nachricht zugegriffen werden kann.

- Die Methode getMessage()

Die Methode public string getMessage() gibt den Nachrichtentyp (z.B. "K001") als String zurück. Diese Methode wird bei einem Recoveryfall vor der Abfrage zur Ausführung der Recoverydatei durch die Klasse Programm aufgerufen, damit der Benutzer weiß welcher Befehl nicht ordnungsgemäß abgearbeitet wurde.

17.4 Programmbeschreibung zenOn-Proxy

- Die execute() Methode

Die execute()Methode der Klasse Commandak1 dient zur Übermittlung von Acknowledge1 beziehungsweise Fehler0. Um erkennen zu können ob die SPS den Befehl ordnungsgemäß verstanden hat oder ein Fehler beim Erkennen der Nachricht aufgetreten ist müssen die zugehörigen Merker der SPS abgefragt werden. Dies erfolgt durch die Klasse MerkerLesen, die am Anfang der execute()-Methode initialisiert wird. Das Auslesen der Merker erfolgt hierbei durch die Methode warteA001() der Klasse MerkerLesen, die einen Booleanwert als Rückgabewert liefert. Dieser nimmt beim ordnungsgemäßen Empfang der Nachricht den Wert true an, so dass der String für Acknowledge1 generiert wird. Wird stattdessen der Merker für den fehlerhaften Empfang der Nachricht gesetzt, liefert die Methode ein false. Im zugehörigen else-Zweig wird anschließend der String für Fehler0 erstellt. Abschließend wird das Ergebniss dieser Bearbeitung mit der Methode sendToServer(acc1, 25) der Klasse Client an die Transportsteuerung gesendet.

17.4.8 Die Commandak2 Klasse

Auch die public class Commandak2:ICommand ist eine abstrakte Klasse der Schnittstelle ICommand. Die Initialisierung erfolgt auch hier analog zur Klasse Command1dat über die CommandFactory. Aufgabe der Klasse ist es die mit Acknowledge2 in verbindungstehende Kommunikation abzuwickeln.

- Der Konstruktor Commandak2

Der Konstruktor internal Commandak2(Message MESSAGE) bekommt ein Objekt MESSAGE der Klasse Message übergeben, damit in der Klasse Commandak2 auf die Variablen der gesendeten Nachricht zugegriffen werden kann.

- Die Methode getMessage()

Die Methode public string getMessage() gibt den Nachrichtentyp (z.B. "K001") als String zurück. Diese Methode wird bei einem Recoveryfall vor der Abfrage zur Ausführung der Recoverydatei durch die Klasse Programm aufgerufen, damit der Benutzer weiß welcher Befehl nicht ordnungsgemäß abgearbeitet wurde.

- Die execute() Methode

Die execute()-Methode der Klasse Commandak2 wickelt die Kommunikation zu Acknowledge2 ab und gibt an ob der gesendete Befehl ordnungsgemäß durchgeführt wurde. Auch hier erfolgt die Bestätigung der Abarbeitung über Merker der SPS, die über die COM-Schnittstelle von Zenon ausgelesen werden können. Bevor dies jedoch möglich ist muss vorab die COM-Schnittstelle am Anfang der execute()-Methode initialisiert werden. Anschließend wird mittels if-Abfrage geprüft ob der Fehler0 bereits aufgetreten ist, da für diesen Fall ein senden von Acknowledge2 nicht weiter erwünscht wäre. Wurde Fehler0 empfangen wird die Erstellung und das Senden von Acknowledge2 übersprungen, andernfalls wird mit der Abfrage der Merker für Acknowledge2 und Fehler1 begonnen. Diese

17.4 Programmbeschreibung zenOn-Proxy

Abfrage erfolgt analog zu Acknowledge1 über die Klasse MerkerLesen. Hierbei wird jedoch auf eine andere Methode warteA002() zugegriffen, die vom Aufbau und der Funktionalität der Methode warteA001() gleicht. Auch hier wird in Abhängigkeit des zurückgegebenen Boolwerts entschieden, welcher String zuerst zusammengebaut und anschließend mit der sendToServer()-Methode an die Transportsteuerung gesendet werden soll. Beim zusammenbauen der Antworten für das Steuersystem besteht jedoch ein Unterschied zu Acknowledge1, da neben dem TCP-Header noch weitere Daten mitgesendet werden können. Beim senden von Acknowledge2 wird nur für den Befehl neuer Schlitten an Fixierstation ((nachricht.Nachrichtentyp) == „K002“ && nachricht.Id3.Equals(-1)) ein Payload angehängt, da die Transportsteuerung eine SchlittenID für diesen neuangeforderten Schlitten benötigt. Alle anderen Befehle beziehen sich auf bereits bekannte Schlitten, so dass ein mitsenden der SchlittenID nicht mehr notwendig ist. Für den Fall Befehl nicht ausgeführt ist jedoch zwangsläufig ein Payload erforderlich, der die Ursache der Störung in Form einer FehlerID nach oben kommuniziert. Diese ID wird auch über die Klasse MerkerLesen von der SPS gelesen, jedoch durch die Methode fehlerid(). Im Anschluß an die Erzeugung der Antwort-Strings erfolgt auch hier das Senden über die sendToServer()-Methode, welche jedoch aufgrund der unterschiedlichen String-Längen variiert.

17.4.9 Die Klasse MerkerLesen

Wie bereits erwähnt, dient die Klasse MerkerLesen zum Auslesen der Merker der SPS. Hierzu wird zwischen drei verschiedenen Fällen unterschieden, für welche folgende Methoden existieren.

- Die Methode warteA001()

Wie bereits in der Erläuterung zu Commandak1 erwähnt dient die Methode public bool warteA001() zum auslesen der Merker für Acknowledge1 beziehungsweise Fehler0. Dieser Vorgang wird solange in einer while-Schleife wiederholt, bis es zu einer Veränderung in einen der beiden Werte kommt. Die Abfrage erfolgt hierbei über die Zenon COM-Schnittstelle, welche am Anfang der Schleife initialisiert wird, damit auch im Falle eines kurzzeitigen Verbindungsverlustes zu Zenon die Kommunikation wieder aufgenommen werden kann. Da bei dieser Konstruktion für jeden Schleifendurchlauf eine neue Instanz erzeugt wird, ist es notwendig die COM-Schnittstelle am Ende der Schleife zu dereferenzieren um den Speicher nicht unnötig zu blockieren. Die Abfrage der Werte erfolgt hierbei in zwei ineinander geschachtelten if-Abfragen. Als Bedingung der ersten if-Schleife wird der Merker für Acknowledge1 abgefragt. Wird dieser durch die SPS auf true gesetzt, wird innerhalb des if-Zweigs das Flag befehlverstanden auf true und die Austiegsbedingung für die while-Schleife gesetzt. Bleibt Acknowledge1 jedoch aus wird innerhalb des else-Zweigs die Zweite if-Abfrage gestartet, welche den Merker für Fehler0 als Bedingung besitzt. Kann der Befehl nicht ordnungsgemäß interpretiert werden, wird die Austiegsbedingung der while-Schleife ohne vorheriges Setzen des befehlverstanden-Flags gegeben. Das befehlverstanden-Flag wird anschließend an die Klasse Commandak1 zurückgegeben. Tritt weder Acknowledge1 noch Fehler0 ein wird die Schleife endlos ausgeführt.

- Die Methode warteA002()

17.4 Programmbeschreibung zenOn-Proxy

Die Methode public bool warteA002() unterscheidet sich von der Methode warteA001() nur durch den Zugriff auf unterschiedliche Merker und Flags. Im Aufbau und in die Funktionsweise unterscheiden sich die beiden Methoden nicht. Der Aufruf von warteA002() erfolgt wie bereits erwähnt in der Klasse Commandak2.

- Die Methode feherid()

Auch die Methode public string fehlerid() arbeitet bei der Kommunikation mit der SPS auf einer Instanz der Zenon COM-Schnittstelle. Zum Auslesen der FehlerID wird hierbei auf die Methode ReadF002FehlerID() zurückgegriffen, welche einen String als Rückgabewert liefert. Dieser String wird anschließend in einer switch-case-Anweisung auf die Art des Fehlers überprüft. Der Zenon-Proxy kann zwischen sechs verschiedenen Fehlern unterschieden, nicht identifizierbare Ids werden defoultmäßig als Sammelfehler behandelt.

17.4.10 Die Klasse COM

Wie bereits in den vorangegangenen Erläuterungen erwähnt dient die Klasse Com als COM-Schnittstelle zu Zenon. Ihre Aufgabe ist es mit Hilfe der im folgenden beschriebenen Methoden die Kommunikation zur SPS abzuwickeln. Grob kann hierbei zwischen schreibenden und lesenden Methoden unterschieden werden.

- Der Konstruktor Com

Am Anfang des Konstruktors internal Com() muss zuerst ein COM Objekt erzeugt werden (app = new zenOn.ApplicationClass()). Anschließend wird durch den Aufruff app.Projects().Item(0) das Zenon Projekt ausgewählt mit dem im folgenden gearbeitet werden soll. Nach der erfolgreichen Konfiguration der COM-Schnittstelle kann daraufhin mit der Initialisierung der Variablen begonnen werden. Hierzu wird den benötigten Variablen des Zenon COM-Objekts die entsprechende Variable des Projekts zugewiesen (K001.'Weiche' stellen = pro.Variables().Item („K001.'Weiche' stellen“)). Der Übergebene String representiert hierbei den Namen, den diese Variable im gewählten Projekt begleitet.

- Die Read Methoden

Die Lesenden Methoden der COM-Schnittstelle fungieren alle nach dem selben Schema. Zuerst wird der Wert des Merkers als Objekt über die get'Value()-Methode des Zenonobjekts herausgelesen. Anschließend wird dieses Objekt unter Verwendung der verschiedenen Methoden der Klasse Convert in den benötigten Datentyp konvertiert (Convert.ToBoolean(value1)). Zum Abschluss wird nun die angepasste Information als Rückgabewert an die aufrufende Klasse übergeben.

- Die Write Methoden

Auch die Methoden zum Schreiben auf die Merker der SPS fungieren alle gleich. Auch hier werden zur Kommunikation Methoden der Zenon-Objektbibliothek verwendet um schreibend auf die Variablen des Zenon-Projekts zugreifen zu können. Bevor jedoch der zuschreibende Wert mit der set'Value()-Methode geschrieben werden kann ist es notwendig den Datentyp des Übergabewerts an die Zenon-Variable anzugeleichen. Hierzu werden wiederum die Methoden der Klasse Convert verwendet. Neben den Übergabewerten wird

17.4 Programmbeschreibung zenOn-Proxy

außerdem noch eine weitere Variable gesetzt, welche anzeigt um welchen Nachrichtentyp es sich bei dem gesendeten Befehl handelt.

- Die Reset() Methode

Die Reset()-Methode ist im eigentlichen Sinn nichts anderes als eine schreibende Methode. Ihre Aufgabe ist es alle Merker der SPS wieder auf ihren Ausgangswert zuschreiben, damit beim Empfang des nächsten Befehls keine Konflikte mit alten Werten auftreten können. Aufgerufen wird die Reset()-Methode nach erfolgreichen Versand von Acknowledgement, da zu diesem Zeitpunkt der aktuelle Befehl vollständig abgearbeitet worden ist. Auch hier erfolgt das Rücksetzen der Variablen mit der set`Value(0, 0)-Methode der Objektbibliothek, jedoch wird anstelle der Übergabeparameter der Wert Null gesetzt.

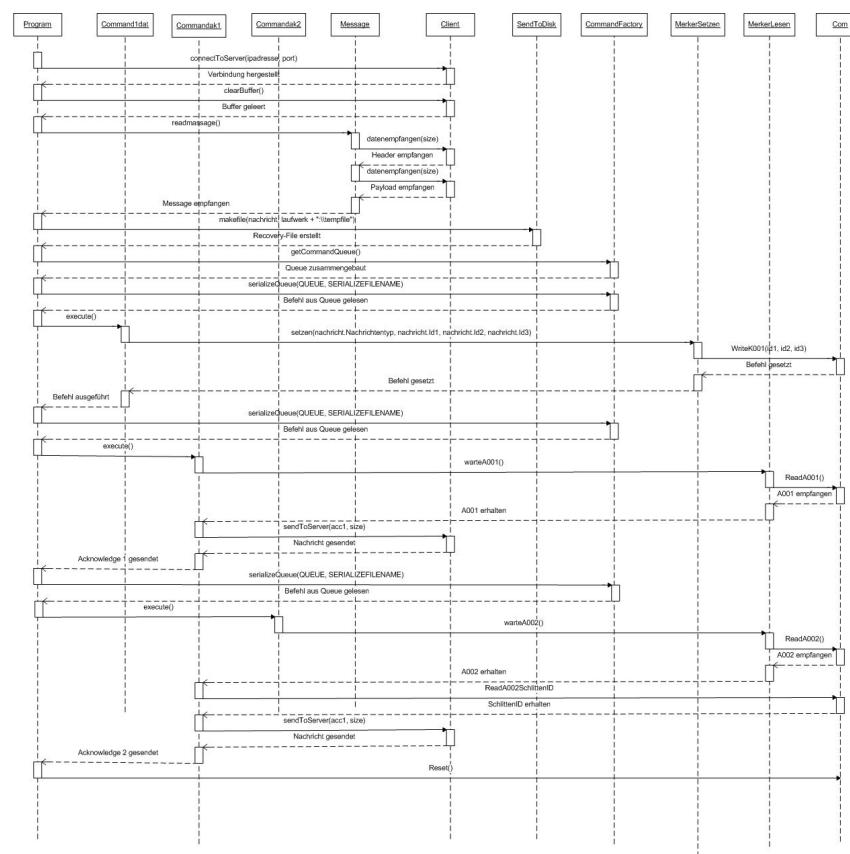


Abbildung 17.2: Sequenzschema des zenOn-Proxy

17.5 Recoverfalltest der Software im IDLE Zustand

Die Software befindet sich im Idle Zustand, wenn der Proxy gestartet wurde, die Verbindung zu Zenon, sowie die Verbindung zum Kommandointerface besteht. Dies bedeutet, dass die Verbindung zum Server mittels der eingegebenen Ip – Adresse / Port aufgebaut wurde, und die Zenon – Referenz mittels der COM korrekt initialisiert werden konnte.

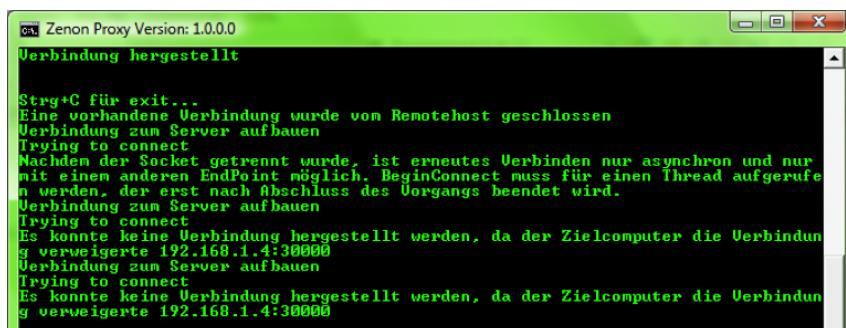
17.5.1 Verlust der Tcp/Ip – Verbindung

Sollte die Tcp/Ip – Verbindung geschlossen werden z.B. durch Verlust der Netzwerkverbindung (physisch oder logisch) so wird zunächst versucht die Verbindung zum Ip – Endpoint wieder aufzubauen. Sollte dies auf Grund des Objektzustands nicht funktionieren, so wird der geöffnete Socket geschlossen und ein neuer Socket initialisiert. Im folgendem wird zunächst wieder versucht eine Verbindung zum Ip – Endpoint aufzubauen sollte dies abermals nicht gelingen wird eine Socket Exception ausgelöst es wird der Socket geschlossen und eine neuer Socket initialisiert. Diese Prozedur wird solange durchlaufen, bis erfolgreich eine Verbindung zum Ip – Endpoint hergestellt werden konnte.

17.5.2 Verlust der Tcp/Ip – Verbindung

Test eines Tcp/Ip Verlust

Der Zenon – Proxy wird zunächst in den bereits beschriebenen Idle – Zustand versetzt. Nun wird die Verbindung durch schließen des Kommandointerfaces getrennt. Dies führt zu folgender Ausgabe.



The screenshot shows a terminal window titled "Zenon Proxy Version: 1.0.0.0". The title bar has a green background. The main area of the window displays the following text:

```
Strg+C für exit...
Eine vorhandene Verbindung wurde vom Remotehost geschlossen
Verbindung zum Server aufbauen
Trying to connect
Nachdem der Socket getrennt wurde, ist erneutes Verbinden nur asynchron und nur
mit einem anderen EndPoint möglich. BeginConnect muss für einen Thread aufgerufe
n werden, der erst nach Abschluss des Vorgangs beendet wird.
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
```

Abbildung 17.3: Versuch des Verbindungsaufbau bei getrennter Leitung

Die Ausgabe wird nun solange Zyklisch wiederholt, bis die Verbindung erfolgreich hergestellt werden konnte.

Sollte dies der Fall sein wird folgendes angezeigt.

17.5 Recoverfalltest der Software im IDLE Zustand

```
Zenon Proxy Version: 1.0.0.0
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Es konnte keine Verbindung hergestellt werden, da der Zielcomputer die Verbindun
g verweigerte 192.168.1.4:30000
Verbindung zum Server aufbauen
Trying to connect
Verbindung hergestellt

Strg+C für exit...
```

Abbildung 17.4: Verbindung wieder hergestellt

Es kann nun wie gewohnt weiter gearbeitet werden.

17.5.3 Recoverfall nach Absturz

Sollte der Proxy eine Nachricht empfangen haben, so wird diese zunächst in eine Datei innerhalb der eigenen Dateien geschrieben. Im folgendem wird aus dieser Message die Queue zusammengesetzt, die empfangene Message gelöscht und die Queue gespeichert. Sollte das Programm geschlossen werden, so muss dieses wieder geöffnet werden. Nachdem das Programm wieder gestartet wurde wird angezeigt, dass ein Recoverfile existiert, welchen Befehl dieses Recoverfile beinhaltet, und ob dieser Befehl geladen werden soll (vgl. Abb.).

17.5 Recoverfalltest der Software im IDLE Zustand

```

Zenon Proxy Version: 1.0.0.0
Bitte die Ip-Adresse des Servers eingeben
Vorschlag: 192.168.1.4 <mit ENTER übernehmen>
Eingabe:
Uorgabe übernommen IP: 192.168.1.4
Bitte den Port des Servers eingeben
Vorschlag: 30000 <mit ENTER übernehmen>
Eingabe:
Uorgabe übernommen PORT: 30000
recovery
Letzter Befehl war: K001
Soll die recover Datei geladen werden??? <J/N>

```

Abbildung 17.5: Abfrage des Recoverfalls

Soll der Befehl geladen so muss vom Benutzer j oder J eingegeben werden. Will der Benutzer den letzten Befehl nicht erneut ausführen so kann er durch Eingabe von n oder N das Recoverfile löschen. Die Software wartet nun auf einen neuen Befehl vom Kommandointerface. Sollte sich der Nutzer für Recovery entscheiden so wird der letzte Befehl entweder komplett abgearbeitet oder vervollständigt.

- Absturz des Proxys beim zusammensetzen der Queue

Zunächst wird im Quelltext ein Haltepunkt in der Klasse CommandFactory beim zusammensetzen der Queue gesetzt um den Absturz zu simulieren.

```

using System.Collections.Generic;
namespace Proxy
{
    class CommandFactory
    {
        static bool deleteflag = false;
        // CommandFactory generiert die passende queue fuer ein gegebenes Szenario
        public static Queue<ICommand> getCommandQueue(String QUEUENAME, Message MESSAGE)
        {
            Queue<ICommand> q = new Queue<ICommand>(); //neue Queue vom Typ ICommand anlegen
            deleteflag = false; //damit später das tempfile nur einmal gelöscht wird
            if (QUEUENAME == Program.QUEUENAME1)
            {
                switch (MESSAGE.Nachrichtentyp) //prüfen welcher Befehl gesendet wurde
                {
                    case "K001":
                        {
                            //Zusammensetzen der Queue für K001
                            q.Enqueue(new Command1dat(MESSAGE));
                            q.Enqueue(new Commandak1(MESSAGE));
                            q.Enqueue(new Commandak2(MESSAGE));
                            break;
                        }
                    case "K002":
                }
            }
        }
    }
}

```

Abbildung 17.6: Haltepunkt setzen

17.5 Recoverfalltest der Software im IDLE Zustand

Es wird nun der Befehl K001 an den Proxy gesendet um den Recoverfall zu testen. Das Programm wird nun abgearbeitet bis der Haltepunkt erreicht ist. Nun wird die Konsole geschlossen. Nach dem Neustart des Programms erhält man folgende Ausgabe.



```
Zenon Proxy Version: 1.0.0.0
Bitte die Ip-Adresse des Servers eingeben
Vorschlag: 192.168.1.4 <mit ENTER übernehmen>
Eingabe:
Vorgabe übernommen IP: 192.168.1.4
Bitte den Port des Servers eingeben
Vorschlag: 30000 <mit ENTER übernehmen>
Eingabe:
Vorgabe übernommen PORT: 30000
recovery
Letzter Befehl war: K001
Soll die recover Datei geladen werden??? <J/N>
```

Abbildung 17.7: Abfrage des Recoverfalls

Es wird angenommen, dass der Benutzer die Recoverdatei laden möchte, dies geschieht durch Auswahl von J oder j. Nun wird der gespeicherte Befehl abgearbeitet. Im Anschluss befindet sich die Software wieder im Idle – Zustand und wartet auf neue Befehle (vgl. Abb.).

17.5 Recoverfalltest der Software im IDLE Zustand



The screenshot shows a terminal window titled "Zenon Proxy Version: 1.0.0.0". The user has typed "j" to load the recover file. The proxy connects to a server and performs a recovery operation. It reads from AC001, writes to AC002, and then sets all values back to zero. Finally, it successfully executes the recovery and waits for further input.

```
zenon Proxy Version: 1.0.0.0
Soll die recover Datei geladen werden??? <J/N>
j
Verbindung zum Server aufbauen
Trying to connect
Verbindung hergestellt

Recoverdatei @: C:\Users\werner\Documents.tempfile
commanddat1
Weiche 1 auf Position0gestellt, SchlittenId:1
K001 gesetzt
WeichenID = 1
Weichenstellung = 0
SchlittenID = 1
Warte auf AC001
A001 lesen
AC001 erhalten
trStA0011246870480:000000
F001 lesen
Warte auf AC002
A002 lesen
AC002 erhalten
K001
trStA0021246870480:000000
commanddak2
Merker zurücksetzen!
Alle Werte auf Null zurück gesetzt!
Nachricht: 0 -----
recover erfolgreich ausgeführt

warte auf Befehl oder leere Buffer

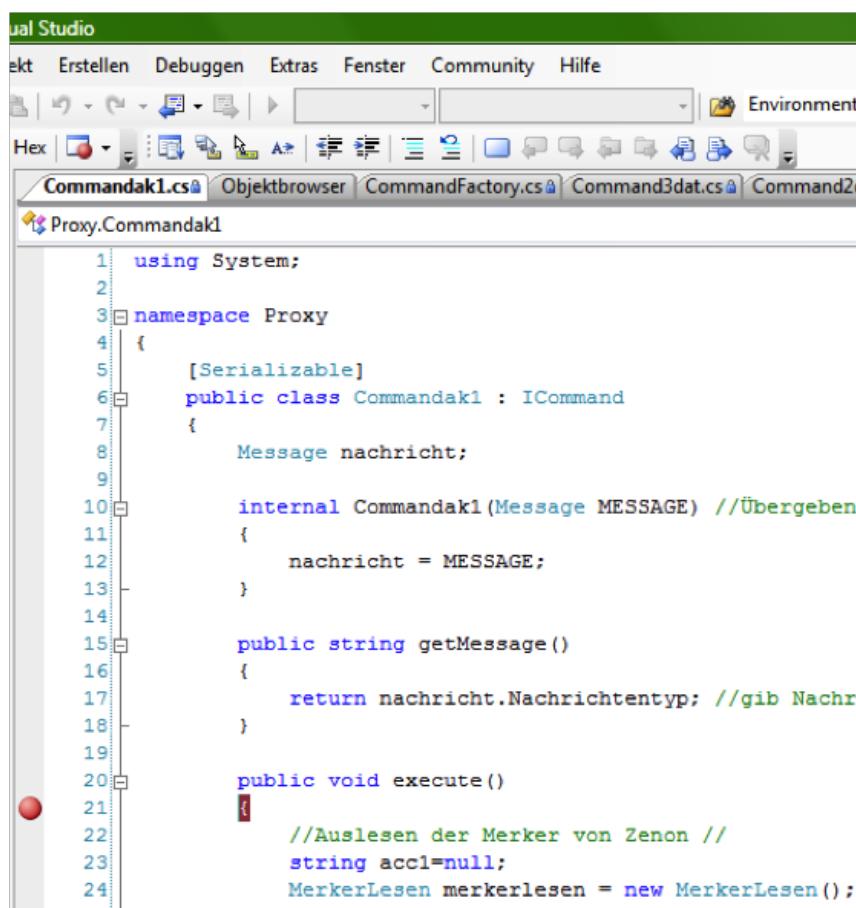
Strg+C für exit...
-
```

Abbildung 17.8: Abarbeiten des Recoverfalls

- Absturz des Proxys beim zusammensetzen der Queue

Zunächst wird im Quelltext ein Haltepunkt in der Klasse Commandak1 beim ausführen des Befehls gesetzt um einen Absturz nach dem zusammensetzen der Queue zu simulieren (Vgl. Abb.).

17.5 Recoverfalltest der Software im IDLE Zustand

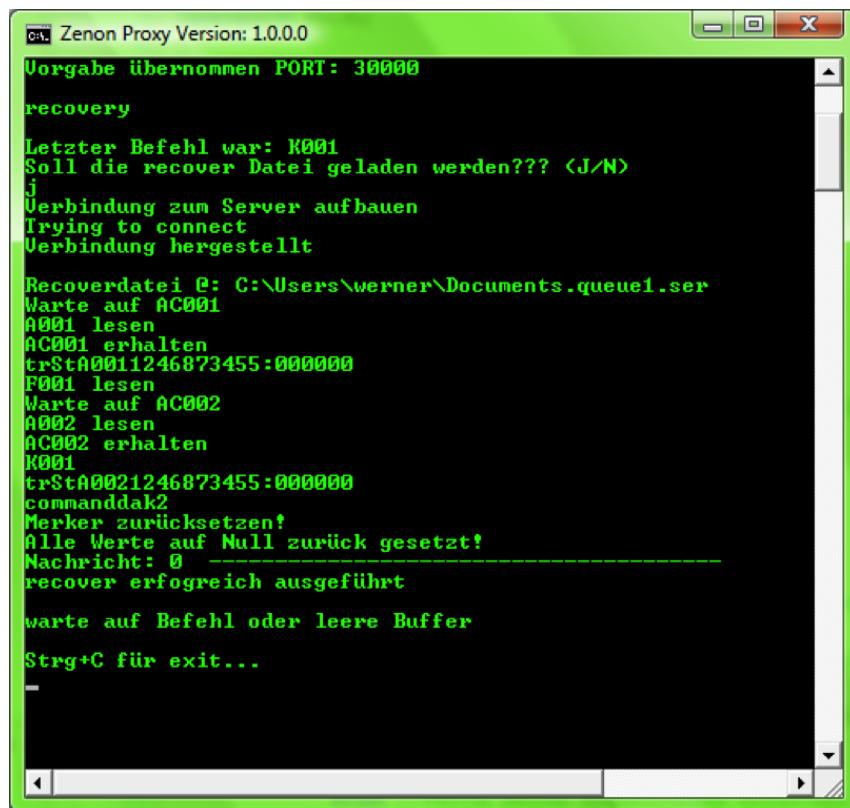


```
1  using System;
2
3  namespace Proxy
4  {
5      [Serializable]
6      public class Commandak1 : ICommand
7      {
8          Message nachricht;
9
10         internal Commandak1(Message MESSAGE) //Übergeben
11         {
12             nachricht = MESSAGE;
13         }
14
15         public string getMessage()
16         {
17             return nachricht.Nachrichtentyp; //gib Nachrichten
18         }
19
20         public void execute()
21         {
22             //Auslesen der Merker von Zenon //
23             string acc1=null;
24             MerkerLesen merkerlesen = new MerkerLesen();
```

Abbildung 17.9: Haltepunkt für simulierten Programmabsturz setzen

Es wird nun der Befehl K001 an den Proxy gesendet um den Recoverfall zu testen. Das Programm wird nun abgearbeitet bis der Haltepunkt erreicht ist. Nun wird die Konsole geschlossen. Nach dem Neustart des Programms erhält man die bereits beschriebene Ausgabe. Auch hier soll der Recovery ausgeführt werden. Im Gegensatz zum ersten Fall wird hier nicht der Komplette Befehl erneut ausgeführt sondern lediglich die für das gespeicherte Kommando verbleibenden Befehle.

17.5 Recoverfalltest der Software im IDLE Zustand



```
Zenon Proxy Version: 1.0.0.0
Vorgabe übernommen PORT: 30000
recovery
Letzter Befehl war: K001
Soll die recover Datei geladen werden??? (J/N)
j
Verbindung zum Server aufbauen
Trying to connect
Verbindung hergestellt
Recoverdatei @: C:\Users\werner\Documents\queue1.ser
Warte auf AC001
A001 lesen
AC001 erhalten
trStA001246873455:000000
P001 lesen
Warte auf AC002
A002 lesen
AC002 erhalten
K001
trStA0021246873455:000000
commanddak2
Merker zurücksetzen!
Alle Werte auf Null zurück gesetzt!
Nachricht: 0 -----
recover erfolgreich ausgeführt
warte auf Befehl oder leere Buffer
Strg+C für exit...
```

Abbildung 17.10: Ausführen der verbleibenden Befehle des gespeicherten Kommandos

Auch hier befindet sich die Software im Anschluss im Idle – Zustand.

17.5.4 Verbindungsverlust an der COM-Schnittstelle

Die Kommunikation zur Bandsteuerung (SPS) erfolgt über die COM-Schnittstelle von Zenon. Für den Fall, dass nach dem Senden eines Befehls keine Verbindung zur SPS hergestellt werden kann, das heißt die Laufzeitumgebung von Zenon geschlossen wurde, ist es vorgesehen auf eine Reaktion der SPS zu warten.

17.5 Recoverfalltest der Software im IDLE Zustand

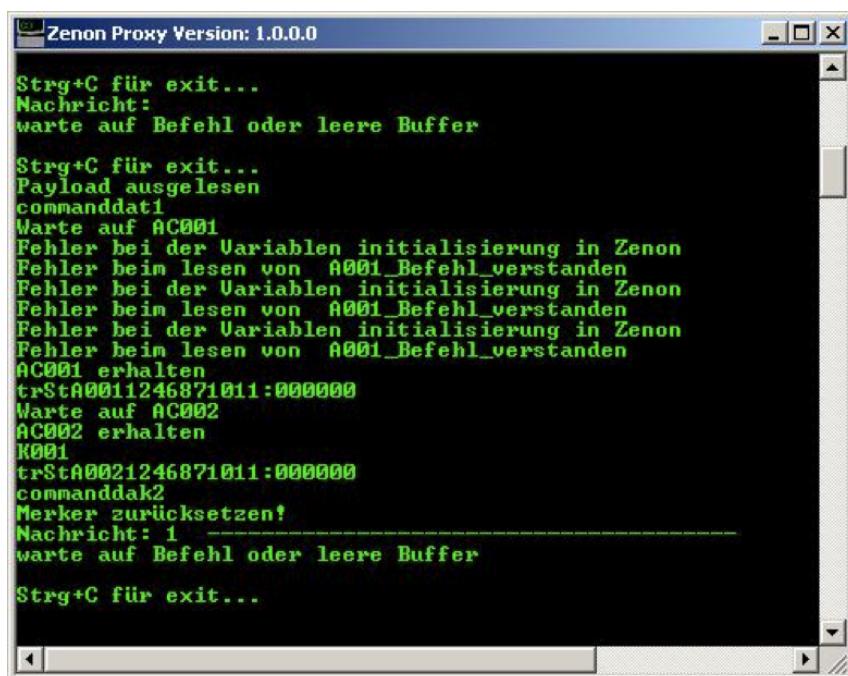


```
Zenon Proxy Version: 1.0.0.0
Strg+C für exit...
Nachricht:
warte auf Befehl oder leere Buffer

Strg+C für exit...
Payload ausgelesen
commanddat1
Warte auf AC001
Fehler bei der Variablen initialisierung in Zenon
Fehler beim lesen von A001_Befehl_verstanden
Fehler bei der Variablen initialisierung in Zenon
Fehler beim lesen von A001_Befehl_verstanden
Fehler bei der Variablen initialisierung in Zenon
```

Abbildung 17.11: Warten auf SPS-Reaktion

Der Zenon-Proxy wartet solange auf eine Antwort der SPS, bis Zenon die Arbeit wieder aufgenommen hat. Da die Abfrage der Antwortparameter solange zyklisch wiederholt wird, bis eine Antwort der Bandsteuerung eintrifft, kann mit der Abarbeitung des Befehls ohne Einschränkungen fortgefahren werden.



```
Zenon Proxy Version: 1.0.0.0
Strg+C für exit...
Nachricht:
warte auf Befehl oder leere Buffer

Strg+C für exit...
Payload ausgelesen
commanddat1
Warte auf AC001
Fehler bei der Variablen initialisierung in Zenon
Fehler beim lesen von A001_Befehl_verstanden
Fehler bei der Variablen initialisierung in Zenon
Fehler beim lesen von A001_Befehl_verstanden
Fehler bei der Variablen initialisierung in Zenon
Fehler beim lesen von A001_Befehl_verstanden
AC001 erhalten
trStA0011246871011:000000
Warte auf AC002
AC002 erhalten
K001
trStA0021246871011:000000
commanddak2
Merker zurücksetzen!
Nachricht: 1 -----
warte auf Befehl oder leere Buffer

Strg+C für exit...
```

Abbildung 17.12: Antwort von SPS erhalten

Teil V

Bekannte Probleme

18 Zum Dokument

- Hyperref Warning: Token not allowed in a PDFDocEncoded string, (hyperref) removing '@ifnextchar' und 'Istinline' liegt an der Definition von 'myFuncRef' in macros.tex.
 - siehe Seite 397 bzw. jniBridgeMessageHandler.tex
 - siehe Seite 393 bzw. jniBridgeMessageHandler.tex

19 Zum Protokoll

- Das Netzwerkprotokoll garantiert NICHT, dass alle gesendeten Nachrichten auch zugestellt werden. Insbesondere kann durch Abbrüche der Netzwerkverbindung zu einem „ungünstigen“ Zeitpunkt im anschliessenden Recoveryfall ein Deadlock auftreten: Das verarbeitende System hat das A002 noch auf eine nicht mehr bestehende Verbindung geschickt. Nach dem Recovery ist das auftraggebende System der Meinung, auf ein A002: „Acknowledge 2“ zu warten, während das verarbeitende System der Meinung ist fertig zu sein, da das A002 ja verschickt wurde.
Eine Lösungsmöglichkeit ist in Schreiben der Log-Dateien beschrieben, jedoch noch in keinem System implementiert.
- Es gibt keine einheitliche Funktion zum Zusammenbau von Nachrichten. Dieses Problem wurde in Implementierung Kommandogenerator und in der *fdzUtility.cpp* aufgezeigt

20 Gemeinsame Module

- In der jniBridge gibt es ein potentielles Memoryleak. Details und Lösung finden sich in Java_jniBridge_fdzMessageHandler_JNIMessageHandler_unregisterJNICallback()
- Prinzipiell ist (unter der Voraussetzung, dass niemals zwei Threads auf ein und dem selben Socket gleichzeitig lesen, klar) die C-Seite der jniBridge Thread-safe, bis auf eine Ausnahme: Das Errohandling kann, wenn auf zwei Threads GLEICHZEITIG Fehler auftreten, eventuell die falsche Klartext-Fehlermeldung zurückgeben. Es werden allerdings BEIDE Fehler erkannt und die Behandlungsroutinen aufgerufen.
vergleiche `jniErrorCallback()` und `jniLastError`

21 Steuerung

- Fehler bei der Nachrichtenübertragung werden nicht berücksichtigt, da davon ausgegangen wird, dass TCP/IP keine fehlerhaften Pakete verschickt.
- Nicht alle Fehlerfälle im Programmablauf können recovert werden. Deshalb wird in diesen Fällen der Programmablauf abgebrochen und der Operator muss in das System eingreifen. Es müssen eventuell die Subsysteme in ihre Ausgangsposition zurückgebracht werden.
- Sollte ein Subsystem eine fehlerhafte Nachricht senden (z.B. 1x7 als Anzahl einer Farbe), wird dies nicht erkannt. Es wird davon ausgegangen, dass die Subsysteme korrekt arbeiten.
- Während der Bearbeitung wird eine Bestandsdatei geführt. Mit deren Hilfe wird die Konsistenz des Lagerbestandes überwacht. Um diese Überwachung zu gewährleisten müsste ständig durch den Operator oder eine Kamera der Lagerbestand aktualisiert werden.
- Alle Konsolenausgaben sollten beobachtet werden, um auf eventuell auftretende Probleme reagieren zu können.

22 Robotersystem

- Generell: Im Praxisblock wurde vereinbart, dass die Produktpalettenposition IMMER 'X' ist, obwohl die Spezifikation zulässt, dass von jeder beliebigen Position zu jeder anderen Position bestückt werden darf. Prinzipiell sollte der Code so geändert werden, dass es möglich ist, von jeder beliebigen Palette zu jeder beliebigen Palette zu bestücken. Der Grund für die Beschränkung der Produktpalettenposition auf 'X' ist, dass der Roboter auf der Position 'X' nicht alle Positionen der Ressourcenpalette erreichen kann (Es können jedoch alle Positionen der Produktpalette erreicht werden). Dies kann sich jedoch jederzeit ändern (neuer Roboter), und dann sollte der Code keine Probleme mit den variablen Positionen haben.
Vergleiche RoboterSteuerungState::getProductPalette.
- Fehlerbehandlung: Schickt ro an Sr ungültige Antworten während der Bestückung (z.B. „roSSA001“), so gibt Sr an ST F000 zurück. Problematisch ist hier jedoch, dass Control dieses F000 nicht von dem Fall „Sr hat den Befehl generell nicht verstanden und noch keine Elementarbefehle an ro verschickt“ unterscheiden kann und auch keinen Überblick über die bereits ausgeführten Bestückungsbefehle hat. Außerdem bleibt aktuell die bestand_assembly.txt Datei bestehen. Hier ist zu klären, was in diesem Fall passieren soll: Soll Sr den Elementarbefehl wiederholen, soll F000 an ST geschickt werden und ST bricht dann generell ab, oder soll eine neue Fehlernachricht (mit welcher Payload?) eingeführt werden?
- Fehlerbehandlung: Eine differenzierte Fehlerbehandlung seitens Sr wäre wünschenswert für den Fall, dass ro einen Befehl mit F000 quittiert.
- Recovery Sr - ro: Wenn der Befehl SrroK* geloggt wurde, roSrA001 aber nicht vorhanden ist, so wird aktuell der Befehl erneut gesendet (war lt. Spec so vorgegeben). Jedoch scheint dies im Falle von Roboterbefehlen wenig sinnvoll: Hat der Roboter den Befehl empfangen, so kann man davon ausgehen, dass das A001 noch vom Netzwerk gelesen werden kann. Also würde ein Warten auf A001 ausreichen. Im ungünstigsten Fall ist Sr abgestürzt, nachdem SrroK geloggt aber bevor es gesendet hat. Dann würde Sr einfach ewig auf roSrA001 warten. Wiederholt man jedoch den Befehl einfach, so läuft man Gefahr, dass der Roboter den ursprünglichen Befehl bereits empfangen und ausgeführt hat, und er den neu empfangenen nochmals ausführt, was zu Doppelbestückungen und ähnlichen gefährlichen Ergebnissen führt.
Beim Recovery ST - Sr ist das erneute Ausführen des Befehls jedoch richtig. Hier wurde der Befehl STSrK* empfangen und geloggt. Wenn SrSTA001 in den Logs fehlt, so wurde die Antwort nicht gesendet und auch keine Roboterbefehle generiert. In dem Fall kann also der STSrK* Befehl einfach erneut ausgeführt werden.
- Generell: Es sollte für die Nachrichtenübertragung und das Recovery eventuell eine objektorientierte Lösung in Betracht gezogen werden, welche vor allem die Weitergabe von

22 Robotersystem

Fehlermeldungen vereinfacht und eine bessere Differenzierung ermöglicht (statt z.B. den globalen Statusvariablen). Auch sollte das Recovery die Logfiles eventuell detaillierter auswerten, vor allem scheint eine unterschiedliche Behandlung von Steuerungsbefehlen und Roboterbefehlen sinnvoll (siehe oben). Da jedoch im WS07/08 das vorhandene Material bereits so vorstrukturiert war, und die Aufgabe aus Bugfixing bestand, war eine Neukonstruktion und -implementierung genannter Teile aus Zeitmangel nicht möglich und auch nicht unsere Zuständigkeit.

23 Lagersystem

24 Transportsystem

- Das Nebenband (Band 2) ist noch nicht implementiert. Die entsprechende Weiche läßt sich zwar schon steuern, jedoch ohne Sicherheitseinstellungen wie z.B. das Prüfen ob der Weichenraum frei ist. Die Kurve im Nebenband ist auch noch nicht geregelt. Die Stopper direkt vor und nach der Fixierstation sowie diese selbst sind noch nicht an das Luftdrucknetz angeschlossen.
- Siehe außerdem ZenOn-Probleme (??).

25 E/A-Station

Keine E/A-Stationsspezifischen Probleme bekannt. Siehe dazu ZenOn-Probleme (??).

26 zenOn-Eigenheiten und -probleme

26.1 Allgemeine Hinweise

Dieser Teil gibt eine allgemeine Anleitung zum Umgang mit zenOn 6.20 SP4, welches zur Entwicklung eingesetzt wurde. Außerdem sollen ferner reproduzierbare Fehler vorgestellt und hinsichtlich ihrer Ursachen und Lösungen erörtert werden.

26.1.1 Beschreibung der eingesetzten Arbeitsumgebung

Eingesetzte Hardware (serverseitig)

Es wird eine SPS von Siemens eingesetzt, welche über einen IBH-Linkadapter ans Netzwerk angebunden ist. *Rechnerpool*

Die Arbeitsstationen in der FDZ sind allesamt mit Windows XP SP2 installiert und bekommen ihre IP Adressen über DHCP. Die Entwickler haben an einem Teil der PCs nur Benutzerrechte, an einem anderen Teil der PCs Hauptbenutzerrechte. Administratorrechte sind nirgendends gewährt worden. Jeder Benutzer hat einen eigenen Account. Dieser Account ist in einer Windows Domäne gespeichert, in welche die Arbeitsstationen aufgenommen sind. Als Entwicklungsumgebung wird das Programm zenon 6.20 SP4 eingesetzt. Dieses Programm ist ein Produkt der Firma COPA-DATA. Die Lizenzierung des Programms erfolgt durch einen zusätzlichen Server mit Win XP, an den ein Dongle angeschlossen ist. Dieser Server ist **nicht** in die Domäne aufgenommen! Auf allen Arbeitsstationen ist zenOn komplett, d.h. incl Datenbankmodul installiert.

26.1.2 Erstellen und Bearbeiten eines zenOn-Projekts

Ein zenOn-Programm teilt sich in Arbeitsbereich und Projekt. Hierbei ist zu beachten, dass der Arbeitsbereich die übergeordnete Ebene ist, welche die Projekte enthält. Hier liegt eine 1:n Beziehung Arbeitsbereich; -> Projekt vor. Auf die einzelnen Bedienelemente von zenOn soll hier nicht eingegangen werden, hierfür ist das Handbuch von zenOn zu konsultieren. Im Folgenden sollen eher die Probleme in der Arbeit mit zenOn dargestellt werden und soweit vorhanden mögliche Lösungsansätze. zenOn bietet die Möglichkeit Bilder einzubinden. Diese Bilder können mit Steuerelementen gefüllt werden, welche später die Variablen anzeigen. Ein Bild muss als Startbild definiert werden. Dieses wird dann defaultmäßig immer angezeigt, wenn die zenOn-Runtime gestartet wird. **ACHTUNG!!** Sollen Änderungen in einem zenOn-Projekt vorgenommen werden, so muss immer zuerst die Runtime beendet werden. Sobald die Runtime sauber komplett geschlossen wurde, werden die gewünschten Änderungen eingepflegt. Jetzt müssen die Runtime-Dateien neu erzeugt werden, bevor die Runtime abermals gestartet wird. Nur so kann sichergestellt werden, dass die Änderungen in der Runtime auch wirklich greifen und zenOn die Änderungen übernimmt. Wird dies nicht so gehandhabt, verhält sich zenOn mitunter nicht

26.1 Allgemeine Hinweise

immer nachvollziehbar und zeigt Werte falsch an oder lädt eine veraltete Projektversion in die Runtime. Wird trotz Befolgens obiger Hinweise immer noch eine veraltete Projektversion in die Runtime geladen, hilft ein kompletter Neustart von zenOn, notfalls ein PC-Neustart, da so die SQL-Engine neu geladen wird. Kompletter Neustart von zenOn bedeutet, dass sowohl die Runtime als auch der Editor komplett geschlossen werden müssen. Nach dem Neustart des Editors sollten die Runtime-Dateien abermals neu erzeugt werden.

Alle eben dargestellten Fälle sind in der Entwicklungszeit mehrfach aufgetreten!

26.1.3 Projektsicherung rücklesen

Durch seine Mehrbenutzerunterstützung muss es in zenOn möglich sein, ein Projekt auf PC1 zu erstellen und dieses Projekt auf PC2 wieder zu laden und dort fehlerfrei auszuführen. Leider war dies sehr oft nicht der Fall. Alle Fallbeispiele liefern wie im Produktivbetrieb nach folgendem Schema ab: Auf PC1 wurde (mit Hauptbenutzerrechten) ein Projekt erstellt. Dieses wurde über die zenOn-Funktion "Projektsicherung erstellen" gesichert. Dieser Punkt befindet sich in der IDE auf der linken Seite relativ weit unten unter "Projektsicherungen". Diese Projektsicherung wird zunächst lokal erstellt. Nach dem erfolgreichen Erstellen wird diese über den Windows-Explorer auf ein Netzlaufwerk kopiert. Auf PC2 wird diese Sicherung vom Netzlaufwerk in den lokalen Ordner *BACKUP* kopiert und von dort über das Menü *Projektsicherung rücklesen* eingelesen. PC2 ist in den meisten Fällen ein PC, an denen die Entwickler nur Benutzerrechte haben, in den seltensten Fällen kann hier mit Hauptbenutzerrechten gearbeitet werden.

Fehlermeldung "Fehler beim Rücklesen der Projektsicherung" Dieser Fehler trat sehr oft (leider nicht willentlich reproduzierbar) auf. Wenn dasselbe Projekt ohne weitere Änderungen danach erneut exportiert und auf PC2 rückgelesen wurde, funktionierte das Rücklesen. *Importproblem nach Umstellung der Service-Pack-Version* Anfangs wurde noch mit unterschiedlichen zenOn-Versionen gearbeitet. Auf einem Teil der Arbeitsstationen wurde anfangs noch zenOn 6.20 SP2 eingesetzt und auf dem anderen Teil zenon 6.20 SP4. Es erscheint logisch, dass man mit SP4 erstellte Projekte nicht in zenOn 6.20 SP2 importierbar sind. Das es anders herum allerdings genausowenig funktioniert, erscheint schon etwas merkwürdig. Beim Rücksichern eines Projekts aus zenOn 6.20 SP2 in zenOn 6.20 SP4 erschien die Fehlermeldung, dass ein Versionskonflikt vorliegt. Infolgedessen musste das Projekt in zenOn 6.20 SP4 nochmal komplett neu erstellt werden.

Korrupte Variablenliste Nach dem Importieren bestand oft das Problem, dass Variablen nicht korrekt importiert wurden, und somit auch nicht mehr ansprechbar waren. Dies ist bei zunehmender Projektgrösse immer häufiger aufgetreten, aber auch leider nicht reproduzierbar. So kam es sehr häufig vor, dass willkürliche Variablen urplötzlich andere Treibereigenschaften hatten. So wurde die Eigenschaft "SPS-Merker" aus unerklärlichen Gründen in "Ausgang" geändert oder die Option "SW aktiv" wurde von zenOn deaktiviert. Ebenfalls aufgetreten ist der Fall, dass nach dem Importieren eines Projekts aus BOOL-Variablen seltsamerweise INT-Variablen geworden sind. zenOn bietet die komfortable Möglichkeit, nur die Variablenliste zu exportieren, was sich in vielen Projektkonstellationen als sehr nützlich erweisen kann. Leider brachte diese Option keine Abhilfe für o.g. Problem. Auch das harte Copy/Paste-Verfahren führte nicht zum Erfolg.

Datenbankfehler beim Starten des Programms Aus unerfindlichen Gründen startete zenOn oft nicht und es kam ein Datenbankfehler. Infolgedessen war es nicht möglich, mit zenOn zu

26.1 Allgemeine Hinweise

arbeiten. Abhilfe schaffte hier meist die Löschung der Inhalte des “SQL“-Ordners. Wenn dies nicht half, wurde zenOn neu installiert.

Mehrfachzugriff auf die Runtime Wenn auf einem PC die zenOn-Runtime läuft, dann sollte es unterlassen werden, von mehreren PCs auf diese Runtime zuzugreifen. Dies könnte konfuse Variablenwerte zur Folge haben wenn zufällig beide zugreifenden Programme einen Wert gleichzeitig versuchen zu ändern.

Dies waren die gröbsten Fehler, die bei zenOn regelmässig auftraten. Kleine vereinzelt nicht reproduzierbare aufgetretene Probleme wurden hier nicht aufgeführt.

Teil VI

Appendix

Abbildungsverzeichnis

Listings